Universidade Federal de Minas Gerais - UFMG

Escola de Engenharia

Manual do Desenvolvedor

Estudante: Arthur Viana Lara
Orientador: Guilherme Vianna Raffo

12 de julho de 2018

Sumário

Lista de Figuras

Capítulo 1

Visão Geral

Veículos aéreos não tripulados (VANTs) são aeronaves equipadas com sistemas embarcados, sensores e atuadores que permitem a realização de voos autônomos ou remotamente controlados. Eles são comumente classificados em dois grupos: veículos de asas rotativas, como helicópteros e quadrotores, e veículos de asas fixas, como aviões.

Há diversas aplicações para VANTs, alguns exemplos são:

- Pulverização de culturas;
- Condução de rebanhos;
- Monitoramento de estradas;
- Inspeção da linhas de energia;
- Entrega de suprimentos em locais de difícil acesso.

Este trabalho está associado ao ProVANT[1]. O ProVANT consiste em uma parceria entre a Universidade Federal de Santa Catarina e a Universidade Federal de Minas Gerais, com o objetivo de realizar pesquisas e desenvolver novas tecnologias para aperfeiçoar o desempenho de VANTs. Neste contexto, atualmente, o ProVANT está focado no desenvolvimento de VANTs Tilt-rotor. O Tilt-rotor está ilustrada na Figura 1.1 e é uma aeronave que possui configuração híbrida, portanto apresenta as principais vantagens das aeronaves de asa fixa e de asa rotativa, como por exemplo consumo reduzido de energia em voos de cruzeiro e decolagem e pouso na vertical. Ele pode operar tanto em ambientes fechados quanto abertos.

Com o intuito de fornecer uma ferramenta de testes para algoritmos e estratégias de controle voltados para vants, desenvolveu-se o simulador ProVANT. Pretende-se, dessa forma, reduzir de custos de projeto e tempo necessário para validação de novas tecnologias.

1.1 Simuladores

Simuladores voltados para aplicações de vants podem ser encontrados de duas maneiras: simuladores de voo e simuladores robóticos.

[1] provant.paginas.ufsc.br

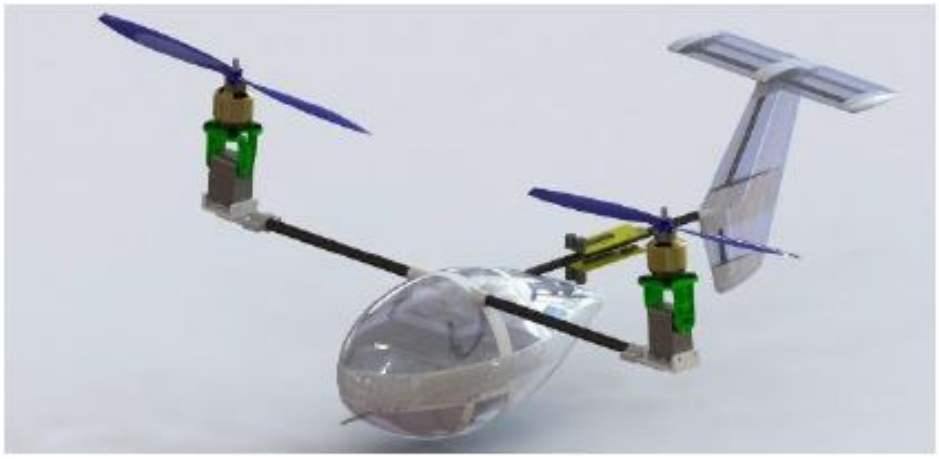

Figura 1.1: Tilt-rotor

Simuladores de voo são ambientes desenvolvidos na maioria das vezes para treinamento de pilotos e/ou para desenvolvimento de jogos de aviões e helicópteros. X-plane e FlightGear são os simuladores mais utilizados por cientistas e profissionais da indústria aeronáutica.

Os simuladores robóticos são ambientes de desenvolvimento de simulação de robôs. O simulador Gazebo e o simulador V-rep são dois exemplos que se destacam nesta classe. Os simuladores robóticos são utilizados para testes e validação de algoritmos de controle e planejamento de trajetória. Estes trabalham com modelos de dinâmica multi-corpo com base nos conceitos de elos e juntas.

A seguir será feito uma revisão bibliográfica sobre aplicações de vants em: i) X-plane; ii) Flightgear; iii) Gazebo; iv) V-rep.

1.1.1 X-plane

O X-plane[2] é um simulador de voo multiplataforma (Linux, MAC e Windows) com a credencial da FAA (Administração Federal de Aviação) que simula voos baseado nos efeitos das forças sobre as múltiplas seções de uma aeronave. O X-plane inclui mais de 30 exemplares de aeronaves disponíveis para simulação e possui a capacidade de customização de cenário e importação de novas aeronaves. Esse simulador permite ao usuário configurar diferentes superfícies de controle aerodinâmico e realizar comunicação externa via protocolo UDP. Além disso, realiza tratamento de diferentes condições atmosféricas de acordo com a altitude. No entanto, o simulador apresenta como desvantagens ser um software proprietário, incapaz de simular múltiplos objetos de simulação no mesmo computador e de realizar simulações com modelos multi-corpos.

Em [**?**] foi modelado e simulado um quadrotor no simulador X-plane. A modelagem baseou-se numa aeronave desenvolvida por um grupo de pesquisa do Instituto Tecnológico da Aeronáutica, sendo utilizado o Matlab/Simulink para realizar o controle automático do quadrotor. Em [**?**] e [**?**], desenvolveu-se um ambiente de simulação para múltiplos VANTs na configuração quadrotor de forma semelhante a [**?**]. No entanto, devido ao elevado esforço computacional exigido pelo simulador, fez-se necessário a

[2] www.x-plane.com

utilização de diversos computadores para a aplicação. Já em [**?**], realizou-se uma simulação *Hardware-in-the-loop* no X-plane de um veículo de asas fixas. Neste trabalho, o comportamento mecânico e aerodinâmico do VANT na configuração avião foi simulado pelo X-plane e o respectivo controlador foi implementado por um sistema embarcado. A comunicação entre o simulador X-plane e o controlador foi realizada através de comunicação serial, sendo que o último recebe do simulador em tempo real os dados dos sensores e, após processamento do controlador, reenvia os sinais de controle.

1.1.2 FlightGear

O FlightGear[3] é um software de simulação de voo multiplataforma de código aberto. É extensível e utiliza três modelos de dinâmica de voo: JSBSim,YASim e UIUC.

JSBSim é um modelo de dinâmica de voo genérico com seis graus de liberdade, onde a aeronave é modelada utilizando um arquivo XML, em que se define as propriedades de massa, aerodinâmica e controle de voo. YASim é um modelo de dinâmica de voo que simula o efeito da corrente de ar nas diferentes partes da aeronave. Por fim, o UIUC é um modelo de dinâmica de voo que engloba em simulação um modelo de aerodinâmica não-linear. Ele resulta em simulações com maior grau de realismo, sobretudo em situações de atitudes extremas, como estol e elevado ângulo ataque.

O simulador possui um banco de modelos e cenários disponíveis (cerca de 20.000 aeroportos reais), é capaz de simular de forma precisa falhas de sistemas e instrumentos aeronáuticos. Porém, é incapaz de emular modelos multi-corpos e de realizar simulações com mais de um modelo concomitantemente.

O FlightGear foi utilizado por [**?**] como simulador para VANTs na configuração avião, e como controlador foi utilizado Matlab/Simulink. Já em [**?**], foi desenvolvido um algoritmo *offline* de geração de trajetórias 4D (tempo, longitude, latitude e altitude) para um VANT de asas fixas, sendo utilizado o simulador FlightGear para validá-lo.

1.1.3 Gazebo

Gazebo[4] é um software livre de simulação robótico desenvolvido pela OSRF (Open Source Robotics Foundation), capaz de simular eficientemente populações de objetos em cenários complexos, sejam eles ambientes externos ou internos.

Esse simulador é apropriado para simulações com múltiplos objetos e detecção de colisão, sendo estes constituídos de um ou mais corpos. Esse sistema possui um banco de sensores considerável (ao todo são 12, contendo a capacidade de criar sensores customizados) e permite ao usuário incluir novos cenários e objetos de simulação através de arquivos XML. Além disso, possibilita que o usuário realize simulações na nuvem, por exemplo em servidores da Amazon.

Algumas simulações de VANTs utilizando o Gazebo são encontradas na literatura. Em [**?**], projetou-se um controlador em cascata utilizando a estratégia de controle não linear *Backstepping* para a malha de controle interna e um controlador PD na malha externa. Ademais, validou-se o controlador projetado utilizando ROS (Robotic

[3] www.flightgear.org
[4] www.gazebosim.org

Operating System) e Gazebo. Já em [?], desenvolveu-se um sistema de visão 3D para helicópteros, o qual foi utilizado numa simulação com o Gazebo.

1.1.4 V-REP

V-rep[5] é um software de simulação robótico multi-plataforma, desenvolvido pela Coppelia Robotics, capaz de simular populações de objetos multi-corpos. É um sistema com interface amigável e suporte a várias linguagens de programação, sendo capaz de tratar colisões entre objetos de simulação, criar novos cenários e importar novos modelos.

Apesar de sua vasta utilização para sistemas robóticos, pouco se encontra sobre o uso do V-REP com VANTs. Em [?], apresentou-se um ambiente de simulação desenvolvido com V-REP e ROS. Esse cenário foi utilizado para sintonia de uma abordagem de controle para um VANT VTOL (acrônimo para *Vertical Take-Off and Landing*, que significa decolagem e aterrissagem vertical) baseada em visão computacional. Foi proposto um algoritmo de controle baseado em redes neurais, adotando um estimador de pose com base no conhecimento da posição do local de decolagem.

1.2 Requisitos de projeto software

No inicio do projeto de desenvolvimento do simulador ProVANT, definiu-se os seguintes requisitos não funcionais:

i) Funcionar sobre sistema operacional Ubuntu/Linux;

ii) Ter licença de software livre;

iii) Ser capaz de realizar tratamento de colisões;

iv) Emular modelos multi-corpos;

v) Incluir modelos de simulação com projeto CAD (projetos realizados com auxílio de computadores).

Quanto aos requisitos funcionais, definiu-se:

i) Interface gráfica para configuração de elementos de simulação;

ii) Conjunto de instrumentos para medição e controle de variáveis físicas;

iii) Controlador a ser configurado pelo usuário.

Tendo em vista os requisitos não funcionais do projeto, os simuladores de voo descritos não são adequados, pois estes não são capazes de trabalhar com modelos de simulação multi-corpos. Já os simuladores robóticos descritos atendem a quase todos esses requisitos. Ambos funcionam no sistema operacional Ubuntu/Linux, são capazes de emular modelos multi-corpos e fazem tratamento de colisões.

Entre os simuladores robóticos, o Gazebo é único exemplar que apresenta licença de software livre e acesso externo aos parâmetros do modelo via arquivo XML, facilitando a criação, modificação e armazenamento de modelos de simulação. Porém a

[5] www.coppeliarobotics.com

característica determinante para a escolha do simulador a ser utilizado foi a existência de suporte pelo Gazebo, no início deste trabalho (Setembro/2015), à atuação de juntas rotativas via conjugado mecânico, o que não ocorria no V-REP. Portanto, devido a essas características, o Gazebo foi o simulador selecionado como base de software de simulação.

1.3 Softwares utilizados

Além do software de simulação Gazebo, utilizou-se outros 2 componentes de software para o desenvolvimento do simulador ProVANT: a plataforma de desenvolvimento QT e o ROS (Robot Operating System). A seguir será dado uma visão geral de cada componente.

QT

Parte do ambiente de simulação ProVANT foi desenvolvido na linguagem C++, utilizando a IDE QtCreator e bibliotecas da plataforma QT versão 5. Qt é uma plataforma de desenvolvimento de aplicações multi-plataforma (hardware e software) com ou sem interface gráfica. O Qt é desenvolvido atualmente pela *The Qt Company* e disponibiliza diversos recursos aos programadores, tais como acesso a banco de dados SQL, análise XML, análise JSON, gerenciamento de threads e suporte de rede.

ROS

Robot Operating System (ROS) é uma plataforma de desenvolvimento de código aberto baseado em Linux para desenvolvimento robótico. O ROS provê funcionalidades que vão desde abstração de hardware e controle em baixo nível de dispositivos até navegação, planejamento de movimento e simulação de alto nível. O sistema foi desenvolvido inicialmente em 2007 no Laboratório de Inteligência Artificial de Stanford e agora é mantido por Willow Garage, um centro de pesquisa em robótica e incubadora.

As aplicações ROS são organizadas em estruturas chamadas Stacks, que agregam os Pacotes (em inglês, Packages) onde se encontram os executáveis, códigos-fonte e bibliotecas. Cada um desses níveis contém um arquivo de manifesto (chamdo "package.xml") responsável pela descrição do conteúdo, facilitando o compartilhamento com a comunidade científica.

O ROS é um sistema descentralizado que permite que aplicações sejam executadas de forma distribuída entre as máquinas. Quando um arquivo executável de um pacote é iniciado, este origina um ou mais nós. Os nós representam processos individuais que se comunicam através dos chamados tópicos. Essa estrutura abstrai um sistema de troca de mensagens assíncrono baseado em sockets. Os nós podem atuar como publicadores em múltiplos tópicos simultaneamente. Enquanto publicadores, um único tipo de dado pode ser postado em um certo tópico, normalmente à uma taxa constante.

O elemento final na estrutura do ROS é o chamado ROS Core (roscore), que é executado em uma única máquina e atua como o DNS (Domain Name System). Cada nó deve receber um identificador ROS Core (Uniform Resource Identifier - URI) para que, durante a execução, os nós possam notificar ao ROS Core que eles existem. Essa

estratégia permite que os nós se comuniquem através de conexões remotas e locais online.

1.4 Organização do manual

O Texto está organizado como a seguir:

1. Este capítulo introduz dá uma visão geral sobre o simulador ProVANT
2. O segundo capítulo apresenta a estrutura do ambiente de simulação ProVANT;
3. O terceiro capítulo organização do projeto de software, descrevendo a localização de arquivos e diretórios;
4. O quarto capítulo descreve detalhes do projeto relativo ao simulador Gazebo. Será abordado sobre modelos, cenários e plugins;
5. O quinto capítulo descreve detalhes do Controlador;
6. O sexto capítulo descreve detalhes da Interface gráfica;
7. O sétimo capítulo descreve como é o processo de inicialização de uma instância de simulação.
8. O apêndice A descreve como se configurar um arquivo "CMakeLists.txt" genérico.
9. O apêndice B descreve como se configurar um arquivo "package.xml" genérico.

Capítulo 2

Estrutura do simulador

A estrutura geral do simulador é ilustrada na Figura 2.1. O ambiente de simulação é constituído de três componentes: i) simulador Gazebo; ii) Controlador; iii) Interface gráfica.

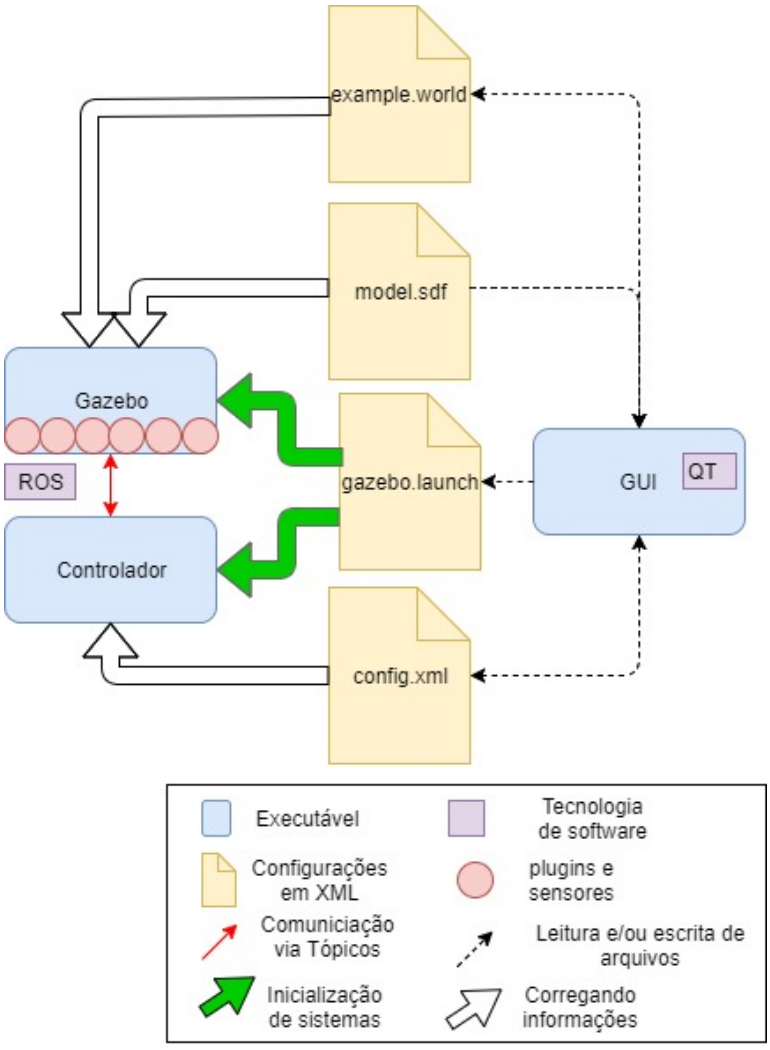

Figura 2.1: Esquemático do simulador

O simulador Gazebo é o componente onde é executado o modelo do vant e, através de plugins e sensores, consegue-se obter e modificar dados durante execução do ambiente de simulação. Para sua configuração inicial, é necessário dois arquivos XML: "model.sdf" e "example.world" (pode-se escolher qualquer nome para este arquivo). Este descreve o cenário de simulação e aquele, o modelo de simulação.

O controlador, por sua vez, é o componente responsável por controlar o voo do vant. As configurações do controlador, tais como período de amostragem e tipo de estratégia de controle, são descritas em um arquivo denominado "config.xml".

Com o intuito de realizar comunicação entre Gazebo e Controlador e automatizar o seus respectivos processos de inicialização, utilizou-se características do ROS. A comunicação entre processos ocorre por meio de tópicos do ROS e a inicialização automática de processos, através de um arquivo XML denominado "gazebo.launch".

Por fim, com a finalidade de tornar amigável o processo de configuração da simulação, desenvolveu-se uma interface gráfica. A interface gráfica é um executável criado utilizando a API do QT, cuja função é ler e/ou editar os arquivos de configuração "model.sdf", "gazebo.launch" e "config.xml" (o sentido as setas na figura 2.1 mostra o fluxo de informação entre o executável e tais arquivos).

Mais detalhes sobre tais elementos serão dados nos próximos capítulos do manual.

Capítulo 3

Organização do projeto de software

O diretório raíz do simulador é chamado *ProVANT-Simulator*, no caso da versão estar no diretório destinado ao público, ou *ProVANT-Simulator_Developer*, para a versão em processo de desenvolvimento. Tal diretório está localizado, após a instalação, no diretório $HOME/catkin_ws/src/. A figura 3.1 ilustra a árvore de diretórios da estrutura do simulador ProVant. No diretório raíz, encontram-se duas pastas: *doc* e *source*: a pasta *doc* contêm a documentação e os manuais de uso; e a pasta *source* contêm o projeto de desenvolvimento da aplicação.

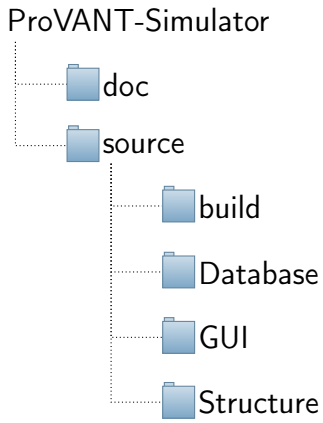

Figura 3.1: Árvore dos diretórios principais do simulador ProVant.

No primeiro nível da hierarquia do diretório raiz do simulador são encontrados os seguintes diretórios:

- *build*: possui os arquivos binários da interface gráfica. É gerado após a instalação completa do ambiente de simulação.
- *Database*: possui arquivos de configuração de modelos, de cenários e da inicialização do ambiente de simulação.
- *GUI*: possui o código fonte da interface gráfica.
- *Structure*: possui o código fonte dos plugins e controlador; arquivos de dados gerados de simulação; e arquivos de descrição das mensagens de tópicos utilizadas para comunicação entre processos no ambiente de simulação.

3.1 Diretório *Database*

O diretório *Database* é responsável por armazenar os cenários para o simulador, os modelos dos VANTs e arquivos de inicialização di ambiente de simulação. A estrutura interna do diretório *Database*, é ilustrado pela figura 3.2.

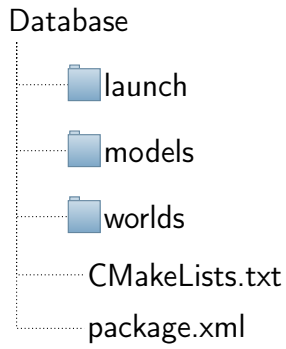

Figura 3.2: Árvore do diretório *Database*.

onde,

- *launch*: possui um arquivo denominado *gazebo.launch*. Este é um arquivo escrito em *XML* que se encarrega dos comandos de configuração do simulador, isto é, inicializa o ambiente do *Gazebo* e executa o nó *controller*.
- *models*: armazena modelos VANTs disponíveis.
- *worlds*: contem os arquivos de extensão *world* correspondentes a descrição de cenários e configurações de simulação.
- *CMakeLists.txt*: armazena informações para compilação de códigos fontes.
- *package.xml*: armazena metadados do diretório, tais como nome e e enereço de e-mail do autor para contato.

3.2 Diretório *GUI*

O diretório *GUI* possui os arquivos *.h* e *.cpp* utilizados na implementação da interface gráfica.

3.3 Diretório *Structure*

O diretório contém o código fonte de todos os componentes do ambiente de simulação, com exceção da interface gráfica. Além disso, é o local para armazenamento de arquivos de dados da simulação. A figura 3.3 ilustra o diretório *Structure*.

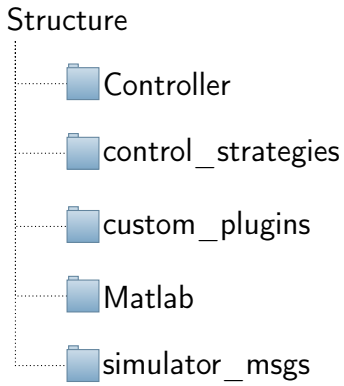

Figura 3.3: Árvore do diretório *Structure*.

onde,

- *Controller*: contem o código fonte do Controlador;
- *control_strategies*:contem o código fonte para diversas estratégias de controle;
- *custom_plugins*: possui o código fonte dos plugins.
- *Matlab*: local onde se armazena arquivos com dados oriundos de simulações.
- *simulator_msgs*: contém arquivos de descrição de mensagens utilizadas para comunicação entre processos via tópicos do ROS.

Capítulo 4

Modelos

4.1 Organização e estrutura

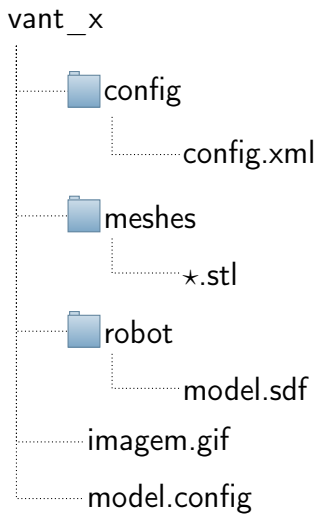

Figura 4.1: Árvore ilustrativa para um diretório que descreve um VANT de versão x *vant_x*.

Os arquivos associados aos modelos de VANTs, utilizados no ambiente de simulação ProVANT, estão localizados na pasta referênte ao caminho:

```
$HOME/catkin_ws/src/ProVANT-Simulator/source/Database/models/real
```

Cada modelo no ambiente de simulação ProVant possui um diretório com o seu respectivo nome. Nesse diretório estão arquivos que descrevem os modelos dinâmicos, visuais, de colisão, sensoriais e da lei de controle utilizada. Portanto, caso seja necessário adicionar um novo modelo, ou editar um existente, o mesmo deve possuir arquivos de configuração/descrição do VANT organizados conforme ilustrado na Figura 4.1. Os diretórios e arquivos necessários são:

- *config*: a pasta *config* possui um único arquivo nomeado *config.xml*. Este arquivo em *XML* escreve configurações do Controlador, tais como período de amostragem

e estratégia de controle a ser utilizada, e será explicitado no capítulo referente a descrição do elemento Controlador.

- *meshes*: a pasta contêm os arquivos de extensão *.stl* que são obtidos no processo de importação do modelo em CAD do VANT pelo *Solidworks*.
- *robot*: a pasta contêm um arquivo *SDF* que descreve o VANT para o simulador.
- *imagem.gif*: corresponde à imagem do vant a ser utilizada na interface gráfica.
- *model.config*: arquivo *XML* com informações sobre o modelo, o nome do desenvolvedor e as licenças de uso.

4.2 Arquivo ”model.sdf”

Antes de apresentar a configuração básica de um arquivo SDF, é necessário primeiro introduzir alguns conceitos. Um modelo corresponde a um sistema mecânico, que pode ser formado por um ou múltiplos corpos rígidos[1]. Assim como em um manipulador, no simulador os corpos são denominados elos. Elos filhos são conectados a elos pai através de juntas. Elos filhos são corpos rígidos que possuem movimento restringido pela conexão ("junta") com corpos denominados elos pai. Os elos possuem propriedades inerciais, visuais e de colisão. Já as juntas, impõem a restrição do movimento relativo entre dois elos, com propriedades como o tipo de junta (Prismática, rotativa, etc.), limites de movimento (Posição e velocidade), existência de atrito, etc. A Figura 4.2 ilustra um exemplo de arquivo SDF, onde:

- <link></link>: especifica a existência de um elo, com o seu nome;
- <joint></joint>: especifica a existência de uma junta, com o seu nome.

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.4">
        <model name="modelo">
                <link name="corpo">
                ...
                </link>
                <link name="servo">
                ...
                </link>
                <joint name="corpo_servo">
                ...
                </joint>
        </model>
</sdf>
```

Figura 4.2: Descrição de um elo no arquivo ”modelo.sdf”

Cada elo do modelo possui três tipos de descrições para o simulador: cinemática, visual e de colisão. A estrutura de configuração de um elo em um arquivo SDF possui o formato ilustrado na Figura 4.3, onde:

[1] Ao assumir um corpo como rígido são desprezando efeitos de elasticidade e deformações

- <pose></pose>: define a pose do elo;
- <inertial></inertial>: especifica as propriedades inerciais do elo;
- <collision></collision>: especifica o modelo de colisão do elo. Os modelos de colisão dos VANTs usados no ambiente de simulação são obtidos de arquivos CAD;
- <visual></visual>: especifica características visuais, como cor e formato. Os modelos visuais dos VANTs usados no ambiente de simulação são obtidos de arquivos CAD, exceto a cor, que é especificada separadamente.

```
<link name="servodir">
        <pose>0.02E-3 -277.61E-3 56.21E-3 -0.0872665 0 0</pose>
        <inertial>
        ...
        </inertial>
        <collision name="servodircollision"> <!--opcional-->
        ...
        </collision>
        <visual name="servodirvisual"> <!--opcional-->
        ...
        </visual>
</link>
```

Figura 4.3: Descrição da de um elo no "modelo.sdf"

Parâmetros de inércia do elo: O usuário deve informar os parâmetros de inércia de cada elo na *tag* "inertial". As informações obrigatórias são a massa, posição relativa do centro de massa e o tensor de inércia. Na Figura 4.4 é ilustrado um exemplo de configuração dos parâmetros de inércia de um elo em formato SDF, onde:

- <mass></mass>: define a massa do elo;
- <pose></pose>: especifica a posição do centro de massa do elo em relação a seu sistema de coordenadas principal;
- <inertia></inertia>: especifica o tensor de inércia do elo;

```
<inertial>
        <mass>0.0809439719362664</mass>
        <pose>
        -3.60859273452335E-10 -0.000226380714807978 0.0594780519701684 0 0 0
        </pose>
        <inertia>
                <ixx>3.88267747087835E-06</ixx>
                <ixy>6.03219085082653E-06</ixy>
                <ixz>-2.78471406661236E-12</ixz>
                <iyy>0.000104858690365283</iyy>
                <iyz>7.0486590219062E-07</iyz>
                <izz>8.31755564684115E-05</izz>
        </inertia>
</inertial>
```

Figura 4.4: Descrição de características inerciais no arquivo "modelo.sdf"

Propriedades de colisão do elo: Para que efeitos de colisão sejam aplicados ao elo, o usuário deve descrever o formato do elo no arquivo "model.sdf". Existem diversas formas de descrição, porém este manual apresenta apenas o método utilizado nos modelos de VANTs do ambiente de simulação ProVANT, que consiste na importação de arquivos criados através de ferramentas CAD, como o SolidWorks. A Figura 4.5 mostra um exemplo de descrição dos parâmetros visuais de um elo a partir de um arquivo STL, onde:

- <pose></pose>: especifica a pose do modelo colisão em relação ao centro de coordenadas do elo;
- <uri></uri>: caminho do arquivo mesh, a partir do diretório do modelo, obtido via exportação no SolidWorks;

```
<collision name="servodircollision">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
                <mesh>
                        <uri>model://vant_2comcarga/meshes/servodir.STL</uri>
                </mesh>
        </geometry>
</collision>
```

Figura 4.5: Descrição de características de colisão no arquivo "model.sdf"

Propriedades visuais do elo: Para que o elo seja visualizado durante a simulação, o usuário deve descrever os parâmetros visuais do elo no arquivo "model.sdf". Assim como no caso anterior, existem diversas formas de descrição, porém este manual ilustra apenas o método utilizado nos modelos de VANTs do ambiente de simulação, que consiste na importação de arquivos criados através de ferramentas CAD. A Figura 4.6 mostra um exemplo de descrição dos parâmetros visuais de um elo a partir de um arquivo STL, onde:

- <pose></pose>: especifica a pose que o modelo visual do elo será definido em relação ao sistema de coordenadas do elo;
- <uri></uri>: caminho do arquivo mesh, a partir do diretório do modelo, obtido via exportação no SolidWorks;
- <ambient></ambient>: definição de cor ambiente;
- <diffuse></diffuse>: definição de cor difusa;
- <specular></specular>: definição de cor especular;
- <emissive></emissive>: definição de cor emissiva.

Descrição de junta

Há 7 tipos de juntas no simulador:

- **revolute**: junta rotativa;
- **gearbox**: junta rotativa com presença de engrenagens para transmissão de movimento angular entre elos com diferentes relações de torque e velocidade;

```
<visual name="servodirvisual">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
                <mesh>
                        <uri>model://vant_2comcarga/meshes/servodir.STL</uri>
                </mesh>
        </geometry>
        <material>
                <ambient>0 0 0 0</ambient>
                <diffuse>1 1 1 1</diffuse>
                <specular>0.1 0.1 0.1 1</specular>
                <emissive>0 0 0 0</emissive>
        </material>
</visual>
```

Figura 4.6: Descrição de características visuais no arquivo "model.sdf"

- **revolute2**: junta composta por duas juntas rotativas em série;
- **prismatic**: junta prismática; (**universal**), junta com comportamento de uma bola articulada;
- **piston**: junta com comportamento da combinação de uma junta rotativa e uma junta prismática.

Um exemplo de estrutura de configuração de uma junta é mostrado na Figura 4.7, onde:

- <pose></pose>: descreve a pose relativa em que o elo filho está em relação ao elo pai.
- <parent></parent>: nome do elo pai.
- <child></child>: nome do elo filho.
- <axis></axis>: vetor unitário que corresponde ao eixo de rotação da junta. (expressado no sistema de coordenadas do modelo, se estiver utilizando a versão 1.4 do formato SDF; e expressado no sistema de coordenadas do elo filho, se estiver utilizando a versão 1.6 do formato SDF).
- <lower></lower>: limite inferior de posição da junta (em rad, caso seja do tipo rotativa, e metros, caso seja prismática).
- <upper></upper>: limite superior de posição da junta (em rad, caso seja do tipo rotativa, e metros, caso seja prismática).
- <velocity></velocity>: limite de velocidade da junta (em rad/s, caso seja do tipo rotativa, e m/s, caso seja prismática).
- <effort></effort>: limite de esforço da junta (em N.m, caso seja do tipo rotativa, e N, caso seja prismática).
- <damping></damping>: coeficiente de atrito viscoso.
- <friction></friction>: coeficiente de atrito estático.

```
<joint name="aR" type="revolute">
        <pose>0 0 0 0 0 0</pose>
        <parent>corpo</parent>
        <child>servodir</child>
        <axis>
                <xyz>0 0.9962 -0.0872</xyz>
                <limit>
                        <lower>-1.5</lower>
                        <upper>1.5</upper>
                        <effort>2</effort>
                        <velocity>0.5</velocity>
                </limit>
                <dynamics>
                        <damping>0</damping>
                        <friction>0</friction>
                </dynamics>
        </axis>
</joint>
```

Figura 4.7: Descrição de juntas no arquivo ”model.sdf”

4.3 Arquivo ”model.config”

No arquivo ”model.config” o gazebo identifica onde está o arquivo com os dados estruturais do modelo, além de informações associadas à autoria, versão e descrição do modelo. A Figura 4.8 ilustra um exemplo desse arquivo. As *tags* utilizadas no arquivo ”model.config” são:

- <name></name>: especifica o nome do modelo;
- <version></version>: especifica a versão;
- <sdf></sdf>: especifica o arquivo com a descrição do modelo dinâmico, de colisão e visual de modelo para o simulador Gazebo;
- <author><name></name></author>: especifica o nome do autor do modelo;
- <author><email></email></author>: especifica o email para contato com o autor;
- <description></description>: descreve brevemente o modelo.

4.4 Arquivos ”meshes”

”Mesh” (do inglês, significa malha) é a representação de objetos virtuais/computacionais em várias áreas da tecnologia, sobretudo na resoluções de problemas de engenharia e é compostas por vértices, arestas e triângulos.

O simulador Gazebo consegue importar dois tipos de arquivos para representação de área e superfícies: arquivos STL e DAE, estes oriundos de softwares CAD. No ambiente de simulação ProVANT, cada modelo vant é formado por vários elos e cada elo terá respectivas representações visuais e de colisão. Por motivos de organização, definiu-se que e o diretório ”Meshes” será o local para armazenamento de tais arquivos.

A seguir será demonstrado um passo a passo de como obter o modelo com seus respectivos arquivos de descrição visual e de colisão.

```
<?xml version="1.0"?>
<model>
<name>vant</name>
<version>1.0</version>
<sdf version='1.5'>robot/model.sdf</sdf>
        <author>
        <name>provant</name>
        <email>provant@ufmg.br</email>
        </author>
                <description>
                        The UAV version 3.0 of the provant project
                </description>
        </model>
</sdf>
```

Figura 4.8: Exemplo de conteúdo existente no arquivo "model.config"

4.5 Obtenção de modelos a partir do SolidWorks

Está subseção descreve o conjunto de passos necessários para a exportação de um projeto mecânico para o formato necessário para uso no simulador Gazebo. Atente-se à necessidade de realizar ajustes no desenho CAD para o processo de exportação tenha sucesso.

4.5.1 Ajustes no Desenho CAD

Para poder exportar o modelo para o formato necessário para uso no simulador Gazebo é necessário realizar o *download* e instalação do plugin "Sw_Urdf_Exporter". O plugin pode ser encontrado em http://wiki.ros.org/sw_urdf_exporter.

Além de *download* e instalação do plugin, é necessário garantir que não exista nenhuma peça com o valor da massa substituída manualmente. Isso gera inconsistências entre o modelo em CAD e o modelo exportado. Uma maneira solucionar este problema, é criar um material personalizado, com uma densidade que corresponda a massa desejada, tal valor é obtido através da relação entre massa e volume.

Outro ajuste importante é a definição de quantos elos deve ter o modelo. Neste manual, exemplifica-se o processo de exportação para o modelo VANT-3.0, ilustrado na Figura 1.1.

O VANT 3.0, ao todo possui sete corpos:

main_body é o corpo referência do VANT-3.0, composto pelo Corpo, suporte dos motores e parte fixa da empenagem;

motorR é o grupo propulsor da direita com exceção da hélice;

motorL é o grupo propulsor da esquerda com exceção da hélice;

propeller_R é a hélice da direita;

propeller_L é a hélice da esquerda;

elevator é a parte móvel da empenagem horizontal;

rudder é a parte móvel da empenagem vertical.

É necessário criar um eixo de coordenadas no centro de massa de cada um dos corpos, para que possam ser referenciados no momento da exportação. No caso do VANT-3.0 são criados eixos de coordenadas sobre o eixo de rotação dos servos. Outro eixo além dos citados é a referência do main_body, localizado onde, teoricamente, pode-se encotrar a IMU. É importante a criação de eixos, que determinam como os corpos se movimentam, já que só temos movimentos de revolução. No VANT-3.0 existem eixos criados no centro de rotação das hélices e nos servos.E também no profundor e no leme.

4.5.2 Processo de Exportação

O processo de exportação de um arquivo ".sdlprt" para ".urdf" é simples e intuitivo. O recurso se encontra em File->Export as URDF, o endereço pode mudar entre versões do SolidWorks®, no caso foi usada a de 2014.

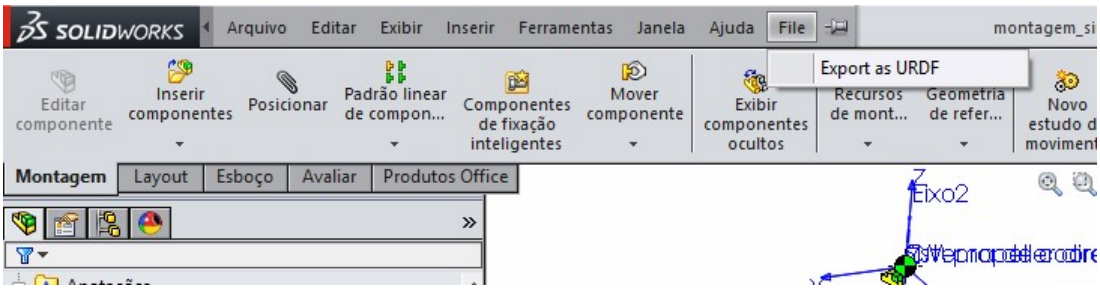

Figura 4.9: Tela onde se inicia o processo

No canto direito da tela abrirá uma tela para inserir os parâmetros. Os primeiros parâmetros a inserir são sobre o link principal que serve de referência para o modelo, no caso do VANT-3.0 o main_body. Os itens a serem detalhados estão listados abaixo:

Link Name é o nome dado ao link principal do modelo a ser exportado;

Global Origin Coordinate System é o eixo de coordenadas que será a referência para esse link e para todo o modelo;

Link Components local onde se define quais componentes do desenho CAD farão parte do link pai;

Number of child links é onde se determina quantos links serão ligados ao link em questão;

Após a definição do link principal, deve-se selecionar um link filho para editar. A seleção é feita no canto inferior esquerdo, clicando-se em Emptyl_link, como mostrado na figura abaixo.

Em seguida deve se editar cada link filho do modelo. Como os links filhos se movimentam, algumas informações adicionais têm que ser inseridas:

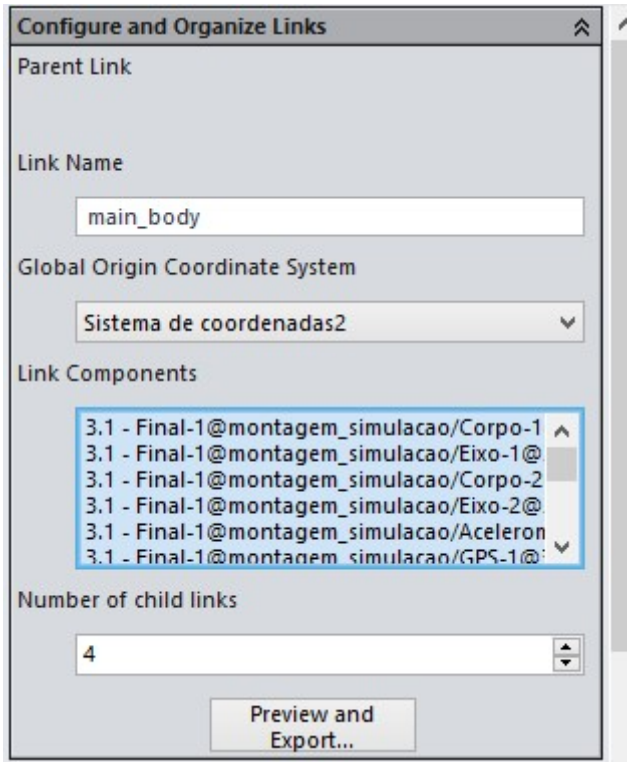

Figura 4.10: Tela para configuração do link principal

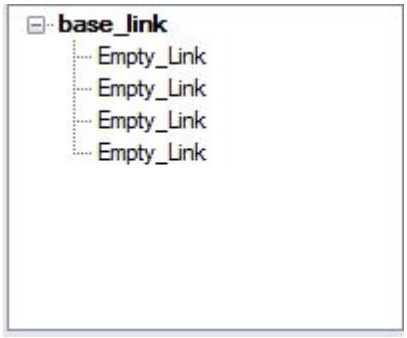

Figura 4.11: Tela para se iniciar a configuração de um link filho

Link Name é o nome dado ao link do modelo a ser exportado;

Joint Name é o nome dado a junta do modelo a ser exportado;

Reference Coordinate System é o eixo de coordenadas que será a referência para esse link;

Reference Axis é o eixo de referência para o movimento da junta;

Joint Type é o tipo da junta, definido pela maneira como a qual se movimenta. Pode ser por revolução, contínua, prismática ou fixa;

Link Components local onde se define quais componentes do desenho CAD farão parte do link especificado;

Number of child links é onde se determina quantos links serão ligados ao ao link em questão;

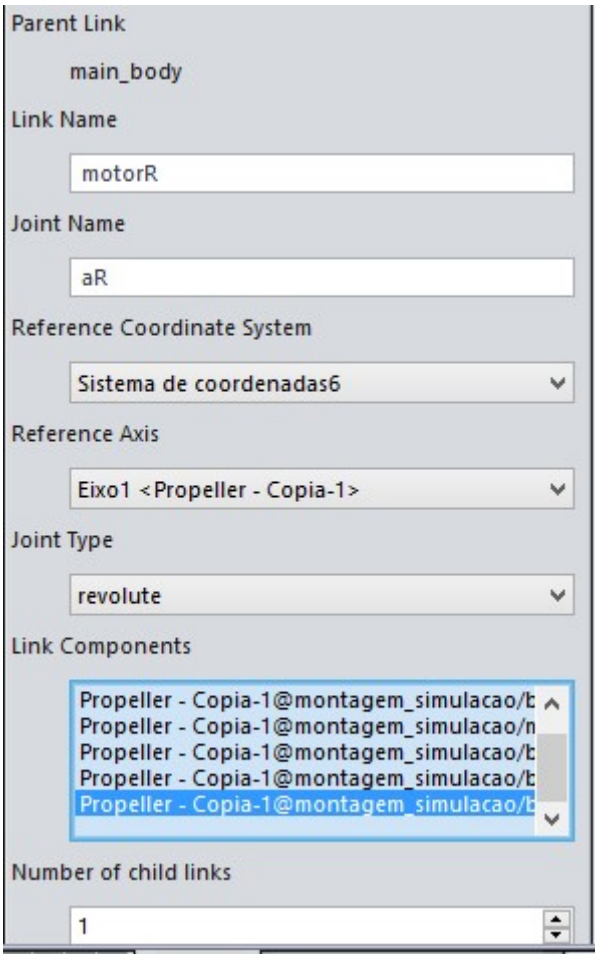

Figura 4.12: Tela para configuração de links filhos

Para o VANT-3.0 foram usadas as configurações encontradas nas Tabelas 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 e 4.7.

Link Name	main_body
Global Origin Coordinate System	Sistemas de coordenadas2
Link Components	Todas as peças da montagem 3.1 - Final, com exceção do profundor e do leme
Number of child links	4

Tabela 4.1: Elo correspondente ao corpo principal

Após a edição dos parâmetros deve-se clicar em 'Preview and Export...' para começar a conversão. Uma tela irá aparecer com os dados das juntas. É importante conferir os valores para assegurar um bom modelo exportado, caso em alguma junta todos os campos aparecerem em branco de ser realizada uma nova exportação. Após a verificação deve-se clicar em 'next' e realizar as mesmas conferência para os links. Não

Link Name	motorR
Joint Name	aR
Reference Coordinate System	Sistema de coordenadas6
Reference Axis	Eixo1<Propeller - Copia-1>
Joint Type	revolute
Link Components	Grupo propulsor da direita sem a hélice
Number of child links	1

Tabela 4.2: Elo correspondente ao servo motor direito

Link Name	propeller_R
Joint Name	thurst
Reference Coordinate System	Sistema de coordenadas6
Reference Axis	Eixo2<Propeller - Copia-1>
Joint Type	continuos
Link Components	Hélice da direita
Number of child links	0

Tabela 4.3: Elo correspondente à helice direita

Link Name	motorL
Joint Name	aL
Reference Coordinate System	Sistema de coordenadas4
Reference Axis	Eixo1<Propeller - Copia-2>
Joint Type	revolute
Link Components	Grupo propulsor da esquerda sem a hélice
Number of child links	1

Tabela 4.4: Elo correspondente ao servo esquerdo

Link Name	popeller_L
Joint Name	thurst
Reference Coordinate System	Sistema de coordenadas4
Reference Axis	Eixo2<Propeller - Copia-2>
Joint Type	continuos
Link Components	Hélice da esquerda
Number of child links	0

Tabela 4.5: Elo correspondente à hélice esquerda

Link Name	elevator
Joint Name	elev
Reference Coordinate System	Sistema de coordenadas9
Reference Axis	Eixo1<3.1 - Final/Montagem1-1>
Joint Type	revolute
Link Components	Profundor
Number of child links	0

Tabela 4.6: Elo correpondente à empenagem horizontal

Joint Name	rud
Reference Coordinate System	Sistema de coordenadas10
Reference Axis	Eixo2<3.1 - Final/Montagem1-1>
Joint Type	revolute
Link Components	Leme
Number of child links	0

Tabela 4.7: Elo correspondente à empenagem vertical

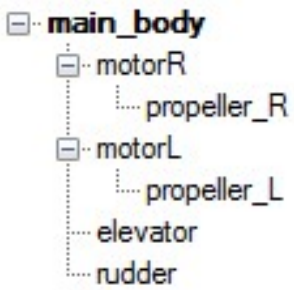

Figura 4.13: Estruturação dos links no VANT-3.0

é recomendável alterar nenhum dado nessa etapa, podendo ocasionar em um modelo que não representa a realidade. Em seguida seleciona-se a opção 'Finish..' e escolhe-se a pasta adequada para salvar. As configurações ficam salvas dentro do SolidWorks® caso seja necessário repetir o processo. zz

4.5.3 Modificações necessárias no projeto exportado

Adicionar arquivo config.xml

Adicione o subdiretório config ao produto resultante do projeto de exportação e crie o arquivo "config.xml" dentro do mesmo. O conteúdo deste arquivo está descrito na seção 7.2

Tradução de arquivos SDF para URDF

O arquivo de descrição cinemática resultante do processo de exportação, que se encontra dentro do diretório *urdf*, é do tipo URDF e o formato melhor adequado para uso no simulador Gazebo é o arquivo SDF.

Antes de realizar o processo de conversão de arquivos, crie o subdiretório "robot", abra um novo terminal com localidade dentro do diretório do produto de exportação e utilize o seguinte comando via terminal (com devidas modificações):

```
gzsdf print urdf/oldgazeboformat.urdf > robot/model.sdf
```

Posteriormente, crie o arquivo "model.config" dentro do diretório do projeto e preencha conforme ilustrado na figura 4.14

```
<?xml version="1.0" ?>
<model>
        <name>vant1gazebo</name>
        <version>1.0</version>
        <sdf version="1.4">robot/model.sdf</sdf>
        <author>
                <name>provant</name>
                <email>provant@hotmail.com</email>
        </author>
        <description></description>
</model>
```

Figura 4.14: Conteúdo a ser colocado no arquivo "model.config"

Capítulo 5

Cenário

5.1 Organização

Os arquivos associados aos cenários, utilizados no ambiente de simulação ProVANT, estão localizados na pasta referênte ao caminho:

```
$HOME/catkin\_ws/src/provant\_simulator/source/Database/worlds/worlds
```

Atualmente, todos os cenários inclusos no ambiente de simulação são cenários vazios. Estes arquivos estão localizados no subdiretório 'Empty' e estão acompanhadas por um arquivo do tipo GIF para sua ilustração na interface gráfica. Estes arquivos configuram o funcionamento do simulador e a inclusão de um único modelo.

Nesta versão á dois cenários: i) Cenário vazio com a inclusão do modelo VANT 2.0; ii) Cenário vazio com a inclusão do modelo VANT 2.0. No entanto o ambiente de simulação não se limita apenas a esses arquivos, a criação de novo cenário é descrita na próxima seção.

5.2 Estrutura

A estrtura básica de um arquivo ".world" está ilustrado na Figura 5.1. Este arquivo é preenchido com tags XML e há 4 comandos principais: i) definição da aceleração da gravidade; ii) definição de configurações do motor de simulação; iii) inclusão do plugin Mundo; iv) inclusão de modelos.

A definição da aceleração da gravidade está na tag <gravity></gravity>. O primeiro elemento corresponde o módulo do componente vetorial na direção x; o segundo, em y; e o terceiro; e o terceiro, em z.

As opções do motor de simulação estão explicitadas na tag <physics></physics>. Internamente, as opções configuradas são: i) **type**, que especifica qual motor de simulação será utilizado; ii) **max_step_size**, que define o valor do passo de simulação; e **real_time_factor**, que define o fator de tempo real que o simulado utilizará. Neste último, se estiver configurada com o valor 0, o passo de amostragem será executado o mais rápido possível; se 1, o passo de simulação será executado conforme o tempo real.

Para incluir um modelo no cenário utiliza-se <include></include>. Neste comando, o campo **uri** especifica o modelo; **name** define o nome do modelo; **static**

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.6">
<world name="vant3.world">
        <gravity>0 0 -9.8</gravity>
        <physics type="simbody">
                <max_step_size>0.001000</max_step_size>
                <real_time_factor>0</real_time_factor>
        </physics>
        <plugin name="gazebo_tutorials" filename="libgazebo_ros_world_plugin.s
        <include>
                <uri>model://ground_plane</uri>
                <static>false</static>
        </include>
        <include>
                <uri>model://sun</uri>
                <static>false</static>
        </include>
        <include>
                <uri>model://vant3</uri>
                <name>newmodel</name>
                <static>false</static>
                <pose>0 0 1 0 0 0</pose>
        </include>
</world>
```

Figura 5.1: Exemplo de cenário

define se o modelo será estático, isto é, o motor de simulação relevará sua existência; e **pose** especifica a pose inicial do modelo.

Por fim, tem-se a inclusão de plugin Mundo por meio da tag <plugin></plugin>. Nela especificamos um nome qualquer por meio de **name** e o nome do arquivo do plugin, **filename**.

5.3 Criando novo cenários

Para criar um novo cenário, exite duas possibilidades: i) adicionar um arquivo cenário já existente (neste caso, cenário vazio), porém com outra configuração de vant; ii) criar um outro cenário.

5.3.1 Adicionar um arquivo cenário já existente

No mesmo diretório do cenário já existente, copie e cole um dos arquivos ".worlds" existentes de maneira a duplicar o mesmo. Modifique o nome do arquivo duplicado para um nome qualquer e altere suas configurações se desejado, por exemplo nome vant e pose inicial.

5.3.2 Criar um outro cenário

Crie um novo diretório. Internamente, crie um arquivo ".world" com as configurações desejadas e adicione um arquivo "imagem.gif" ilustrando esse cenário. Esta foto servirá para uso na interface de auxílio ao usuário.

Capítulo 6

Plugins e sensores

O funcionamento de um vant necessita de intrumentação para medição e atuação de variáveis físicas, por exemplo, GPS e motores. Dada essa importância, o simulador Gazebo fornece ao usuário elementos de simulação capazes ler/modificar variáveis de simulação durante a sua execução.

Esta versão do ambiente de simulação ProVANT fornece um conjunto fixo destes elementos. Assim, com o objetivo de documentá-los e orientar, futura manutenção e desenvolvimento do ambiente de simulação, este capítulo descreve a organização e estrutura da instrumentação.

Há dois tipos de instrumentos na plataforma de simulação: i) elementos padronizados disponíveis o Gazebo de pronto para uso; ii) elementos customizados. Aqueles são denominados "Sensores" e o estes, " Plugins".

6.1 Plugins

Plugins são bibliotecas dinâmicas criadas pelo usuário que são carregadas durante a inicialização do simulador Gazebo a partir das configurações aplicadas no arquivo de descrição de modelo ou no arquivo de descrição de cenário (arquivos SDF). Tais bibliotecas tem a capacidade de interagir com a simulação em execução, seja adquirindo dados e aplicando sinais de controle, seja alterando configurações de simulação.

De modo geral, o Gazebo possui 6 tipos de plugins:

1. Mundo
2. Modelo
3. Sensor
4. Sistema
5. Visual
6. GUI

Dentre o 6 tipos, o ambiente de simulação apenas utiliza os plugins Mundo e Modelo. O diretório com código fonte e arquivos para compilação e construção destes está em:

```
$HOME/catkin\_ ws/src/provant_simulator/source/Structure/
custom_plugins
```

No entando, o diretório com todo o código fonte está organizado no subdiretório **plugins**, conforme ilustrado na figura 6.1.

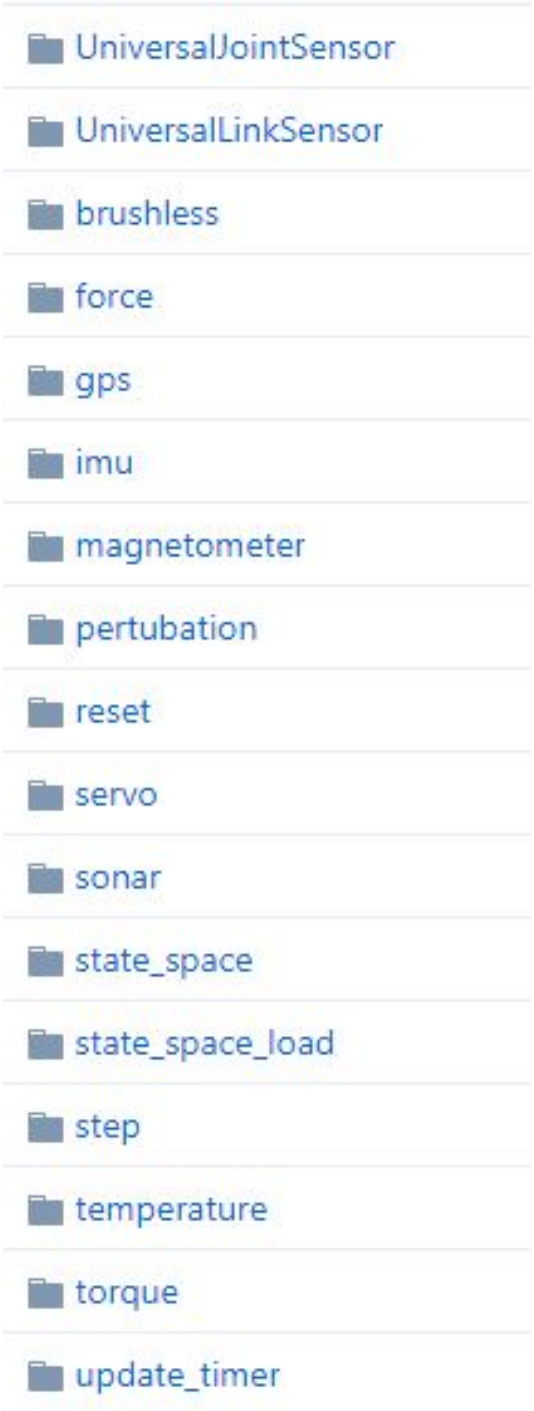

Figura 6.1: Lista de plugins

Mais detalhes leia este tutorial encontrado no site do Gazebo: http://gazebosim.org/tutorials/?tut=plugins_hello_world

6.1.1 Plugins Modelo

Opções disponibilizadas no ambiente de simulação

- Brushless

Descrição: plugin para simulação das forças de empuxo resultado do giro das duas hélices pelos motores brushless de um tilt-rotor

Arquivo: libgazebo_ros_brushless_plugin.so

Configurações:
* <topic_FR> </topic_FR> nome do tópico refente ao valor da força a ser aplicada na hélice direita
* <topic_FL> </topic_FL> nome do tópico refente ao valor da força a ser aplicada na hélice esquerda
* <topic_FL> </topic_FL>
* <LinkDir> </LinkDir> Elo correspondente à hélice direita (a força será aplicada no eixo z desse elo)
* <LinkEsq> </LinkEsq> Elo correspondente à hélice esquerda (a força será aplicada no eixo z desse elo)

- Servo

Descrição: plugin para simulação servo motor com funções de Torque e posição

Arquivo: libgazebo_servo_motor_plugin.so

Configurações
* <NameOfJoint> </NameOfJoint> nome da junta a ser controlada pelo servo motor
* <TopicSubscriber> </TopicSubscriber> nome do tópico com valores de referência para o servo motor
* <TopicSubscriber> </TopicSubscriber>
* <LinkDir> </LinkDir> Nome do tópico com valores de sensoriamento do servo (posição e velocidade)
* <Modo> </Modo> Modo de funcionamento do servo motor

- State_space

Descrição: plugin para sensoriamento do vetor de estados de um VANT Tilt-rotor. $(x, y, z, \phi, \theta, \psi, \alpha_R, \alpha_L, \frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt}, \frac{d\phi}{dt}, \frac{d\theta}{dt}, \frac{d\psi}{dt}, \frac{d\alpha_R}{dt}, \frac{d\alpha_L}{dt})$

Arquivo: libgazebo_AllData_plugin.so

Configurações
* <NameOfTopic> </NameOfTopic> nome do tópico para o usuário obter informações
* <NameOfJointR> </NameOfJointR> nome a junta do servo motor direito
* <NameOfJointL> </NameOfJointL> nome a junta do servo motor esquerdo
* <bodyname> </bodyname> nome do elo correspondente ao corpo principal do servo motor

- State_space_load

Descrição: plugin para sensoriamento do vetor de estados de um VANT Tilt-rotor com a função transporte de carga.
$(x, y, z, \phi, \theta, \psi, \alpha_R, \alpha_L, \lambda_x, \lambda_y \frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt}, \frac{d\phi}{dt}, \frac{d\theta}{dt}, \frac{d\psi}{dt}, \frac{d\alpha_R}{dt}, \frac{d\alpha_L}{dt}, \frac{d\lambda_x}{dt}, \frac{d\lambda_y}{dt})$

Arquivo: libgazebo_AllData2_plugin.so

Configurações
* <NameOfTopic> </NameOfTopic> nome do tópico para o usuário obter informações
* <NameOfJointR> </NameOfJointR> nome a junta do servo motor direito
* <NameOfJointL> </NameOfJointL> nome a junta do servo motor esquerdo
* <NameOfJoint_X> </NameOfJoint_X> nome a junta correspondente ao grau de liberdade da carga em torno do eixo X
* <NameOfJoint_Y> </NameOfJoint_Y> nome a junta correspondente ao grau de liberdade da carga em torno do eixo Y
* <bodyname> </bodyname> nome do elo correspondente ao corpo principal do servo motor

- temperature

Descrição: plugin para sensoriamento da temperatura e pressão atmosférica com ruído.

Arquivo: libgazebo_ros_temperature.so

Configurações
* <Topic> </Topic> nome do tópico para o usuário obter dados sensoriais
* <TempOffset> </TempOffset> offset de erro para dados de temperatura ruidosos
* <TempStandardDeviation> </TempStandardDeviation> desvio padrão de erro para dados de temperatura ruidosos
* <BaroOffset> </BaroOffset> offset de erro para dados de pressão ruidosos
* <BaroStandardDeviation> </BaroStandardDeviation> desvio padrão de erro para dados de pressão ruidosos
* <maxtemp> </maxtemp> valor máximo de temperatura
* <mintemp> </mintemp> valor mínimo de temperatura
* <maxbaro> </maxbaro> valor máximo de pressão
* <minbaro> </minbaro> valor mínimo de pressão
* <Nbits> </Nbits> quantidade de bits utilizados na digitalização

- UniversalJointSensor

Descrição: plugin para sensoriamento de todos os dados que o Gazebo disponibiliza de uma junta. (ângulo,velocidade angular e Torque)

Arquivo: libgazebo_ros_universaljoint.so

Configurações
* <NameOfTopic> </NameOfTopic> nome do tópico para o usuário obter dados sensoriais
* <NameOfJoint> </NameOfJoint> nome da junta para sensoriamento
* <Axis> </Axis> Eixo de rotação da junta ("para primeira junta e "axis2"para segunda junta - gazebo contém juntas q permite dois graus de liberdade)

- UniversalLinkSensor

Descrição: plugin para sensoriamento de todos os dados que o Gazebo disponibiliza de um elo.

Ordem de informações
* pose relativa em x
* pose relativa em y
* pose relativa em z
* pose relativa em ϕ
* pose relativa em θ
* pose relativa em ψ
* velocidade relativa em x
* velocidade relativa em y
* velocidade relativa em z
* aceleração linear relativa em x
* aceleração linear relativa em y
* aceleração linear relativa em z
* força relativa em x
* força relativa em y
* força relativa em z
* velocidade angular relativa em x
* velocidade angular relativa em y
* velocidade angular relativa em z
* aceleração angular relativa em x
* aceleração angular relativa em y
* aceleração angular relativa em z
* conjugado mecânico relativa em x
* conjugado mecânico relativa em y
* conjugado mecânico relativa em z
* pose global em x
* pose global em y
* pose global em z
* pose global em ϕ
* pose global em θ

* pose global em ψ
* velocidade global em x
* velocidade global em y
* velocidade global em z
* aceleração linear global em x
* aceleração linear global em y
* aceleração linear global em z
* força global em x
* força global em y
* força global em z
* velocidade angular global em x
* velocidade angular global em y
* velocidade angular global em z
* aceleração angular global em x
* aceleração angular global em y
* aceleração angular global em z
* conjugado mecânico global em x
* conjugado mecânico global em y
* conjugado mecânico global em z
* velocidade linear do centro de gravidade global em x
* velocidade linear do centro de gravidade global em y
* velocidade linear do centro de gravidade global em z
* pose linear do centro de gravidade global em x
* pose linear do centro de gravidade global em y
* pose linear do centro de gravidade global em z

Arquivo: libgazebo_ros_universallink.so

Configurações
* <NameOfTopic> </NameOfTopic> nome do tópico para o usuário obter dados sensoriais
* <NameOfLink> </NameOfLink> nome da elo para sensoriamento

Inserindo no arquivo de descrição do modelo

Para inserir plugins modelos basta inserir tags <plugin></plugin>, definindo nome, nome da biblioteca dinâmica e as tags internas necessárias para configuração do plugin. O exemplo abaixo demonstra como referenciar plugins modelo no arquivo SDF.

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.4">
        <model name="modelo">
                <link name="corpo">
                        ...
                </link>
```

```
                <link name="servo">
                        ...
                </link>
                <joint name="corpo_servo">
                        ...
                </joint>
                <plugin name="1" filename="1.so">
                        <config1> </config1>
                        <config2> </config2>
                        ...
                </plugin>
                <plugin name="2" filename="2.so">
                        <config3> </config3>
                        <config4> </config4>
                        ...
                </plugin>
        </model>
</sdf>
```

Processo de criação de novo plugin Modelo

O processo de criação de um plugin do ponto de vista de desenvolvimento de software consiste na criação de uma classe (composta por arquivos .hpp e .cpp) com interface definida préviamente. No caso, o código fonte deve herdar a interface que está definida na classe ModelPlugin. Esta classe possui três métodos virtuais que são:

- Init(): método referente ao comportamento personalizado de inicialização do plugin.
- Load(): Método onde carrega as configurações definidas no arquivo XML, sobretudo ponteiros para acesso a manipulação e sensoriamento de elos e juntas.
- Reset(): Método para o comportamento personalizado de redefinição do plugin.

Mais detalhes na classe pai ModelPlugin podem ser encontrados em http://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1ModelPlugin.html

A seguir apresenta o exemplo de um código fonte de um plugin model. Neste exemplo é demonstrado a estruturade um arquivo fonte de um plugin e mostrar como os plugins são interrompidos a todo passo de simulação. Detalhes de como programar cada método com o comportamento desejado pode ser utilizado o código fonte deste projeto e o site http://osrf-distributions.s3.amazonaws.com/gazebo/api/

Arquivo Exemplo.hpp

```
#include <gazebo/physics/physics.hh>
#include <gazebo/transport/TransportTypes.hh>
#include <gazebo/common/Time.hh>
```

```
#include <gazebo/common/Plugin.hh>
#include <gazebo/common/Events.hh>
#include <update_timer.h>

namespace gazebo
{
        class Exemplo : public ModelPlugin
        {
                public:
                virtual void Init();
                virtual void Load(physics::ModelPtr _model, sdf::ElementPtr _sdf);
                virtual void Reset();

                protected:
                virtual void Update();

                private:
                physics::WorldPtr world;
                UpdateTimer updateTimer;
                event::ConnectionPtr updateConnection;
        };
}
```

Arquivo Exemplo.cpp

```
#include <Exemplo.hpp>

namespace gazebo
{
        void Exemplo::Init()
        {
                // Do something when the plugins starts
        }

        void Exemplo::Load(physics::ModelPtr _model, sdf::ElementPtr _sdf)
        {
                // Loading data
                world = _model->GetWorld();
                updateTimer.Load(world, _sdf);
                updateConnection = updateTimer.Connect(boost::bind(&Exemplo::Update, t
        }

        void ServoMotorPlugin::Reset()
        {
                // Do something when reset
        }
```

```
        void ServoMotorPlugin::Update()
        {
                // Do some something in each simulation time

        }

        GZ_REGISTER_MODEL_PLUGIN(Exemplo)
}
```

Os métodos *Init*, *Load* e *Reset*, como já explicado, realizam, respectivamente comandos de inicilização, leitura de informações do arquivo SDF e reset de simulação. Já o método *Update*, não se encontra na interface do plugin pai *ModelPlugin*, mas é um método que é configurado para ser chamado a todo momento que ocorrer um passo de simulação do Gazebo.

Essa configuração ocorre dentro do método *Load*. Este obtém o ponteiro *_model* para dados do arquivo SDF do modelo e através do método do conteúdo deste ponteiro *GetWorld*, ele obtém o ponteiro para a estrtura de dados com as informações do cenário. Posteriormente, este estrutura é carregada na classe *UpdateTimer* e é realizado a sua inicialização. Esta estrtura que informa a necessidade de ocorrências de interrupções a cada passos de simulação.

6.1.2 Plugins Mundo

Um preocupação neste trabalho foi garantir a sincronização entre Controlador (que será descrito posteriormente) e a simulação, conforme ilustrado na figura 6.2. De forma sequencial os subsistemas serão executados e quando um sistema está executando os outros estão em repouso. Iniciamente, acontece um passo de simulação no Gazebo, os sensores obtém dados do ambiente e manda-os para tópicos de sensores. Em seguida os controladores serão avisados que chegou dados nestes tópicos, assim computará sua lei de controle e enviará os sinais de controle obtidos para tópicos de conexão com atuadores. Por fim os atuadorestrasnmitirão os sinais de controle para o amabiente de simulação.

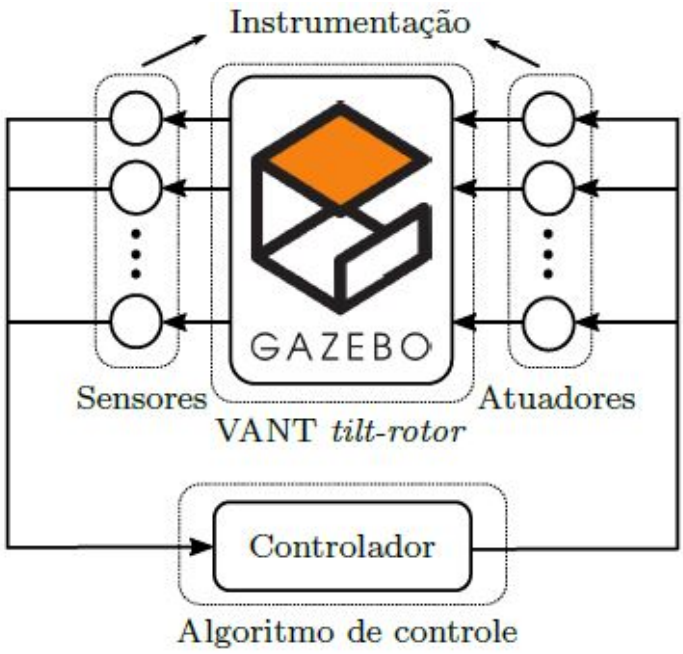

Figura 6.2: Esquemático do simulador

O controlador é um executável que somente é executado quando chegar novos dados de sensores, no entanto, o Gazebo já é configurado por padrão de rodar indefinidamente. Então com a finalidade de garantir que os componentes do sistem executem sequencialmente, propou-se em deixar o simulador em suspensão e quando necessitar o controlador solicita a execução de novo passo de simulação. Para possibilitar a comunicação do controlador e simulador nesse sentido, projetou-se um plugin mundo que através de uma conxão externa através de tópicos do ROS possibilita que o controlador comande a execução do momento necessário para a execução de novo passo de simulação.

A configuração do plugin mundo é definida como:

- Step plugin

Descrição: plugin comando externo do Controlador para solicitação de novo passo de simulação

Arquivo: libgazebo_ros_world_plugin.so

Configuração: Nome do tópico definido permanentemente como Step

6.2 Sensores disponíveis

O sensores, diferentemente dos plugins modelo, são implementados pelo próprio Gazebo e, após serem referenciados no arquivo de descrição do modelo, disponibilizam os seus dados por meio de tópicos do Gazebo. Estes tópicos funcionam semelhantemente aos tópicos do Gazebo, porém possui API própria para lidar com a escrita e leitura de dados. Veja mais detalhes em http://www.robopgmr.com/?p=5614.

No entanto, como os tópicos do ROS e GAzebo não possuem conexão, é necessário fazer trasnmissão de dados entre ambas as tecnologias de comunicação entre processos. Essa comunicação foi feita, desenvolvendo plugins modelos apenas com essa função. Aqui chamaremos de conversores de dados.

Opções disponibilizadas

Diversos são os sensores disponibilizados pelo Gazebo e sua configuração no arquivo SDF estão descritas em http://sdformat.org/spec?ver=1.6&elem=sensor. Porém apenas alguns coversores de dados foram desenvolvidos para esta versão e são GPS, Sonar, Magnetômetro, IMU. Suas configurações são:

- GPS

Descrição: plugin para conversão e transporte de dados entre plugins Gazebo e ROS
Arquivo: libgazebo_ros_sonar.so
Configurações:
* <gazebotopic> </gazebotopic> nome do tópico Gazebo
* <rostopic> </rostopic> nome do tópico ROS
* <link> </link> nome do elo que o sensor está acoplado

- Sonar

Descrição: plugin para conversão e transporte de dados entre plugins Gazebo e ROS
Arquivo: libgazebo_ros_sonar.so
Configurações:
* <gazebotopic> </gazebotopic> nome do tópico Gazebo
* <rostopic> </rostopic> nome do tópico ROS
* <link> </link> nome do elo que o sensor está acoplado

- Magnetômetro

Descrição: plugin para conversão e transporte de dados entre plugins Gazebo e ROS
Arquivo: libgazebo_ros_magnetometer.so
Configurações:
* <gazebotopic> </gazebotopic> nome do tópico Gazebo
* <rostopic> </rostopic> nome do tópico ROS
* <link> </link> nome do elo que o sensor está acoplado

- IMU

Descrição: plugin para conversão e transporte de dados entre plugins Gazebo e ROS
Arquivo: libgazebo_ros_imu.so
Configurações:
* <gazebotopic> </gazebotopic> nome do tópico Gazebo
* <rostopic> </rostopic> nome do tópico ROS
* <link> </link> nome do elo que o sensor está acoplado

Inserindo Sensor e conversor no arquivo de descrição de modelo

Nesta parte será ilustrado como inserir um sensor com seu respectivo conversor de dados no arquivo modelo.

Capítulo 7

Controlador

Este capítulo descreve o Controlador. Este é um software projetado para controlar a simulação garantindo sincronia com o Gazebo. Inicialmente, é descrito a localização de seus componente no projeto e a arquitetura de software utilizada.e seus arquivos de configuração. Em seguida, será exposto como é realizado a sua configuração e o processo de comunicação com os plugins do Gazebo. Por fim, é especificado os plugins de controle existentes, o processo de criação de novos plugins de controle e o processo de armazenamento de dados de simulação.

7.1 Localização no projeto

Controller

A figura 7.1 ilustra o diretório *Controller*.

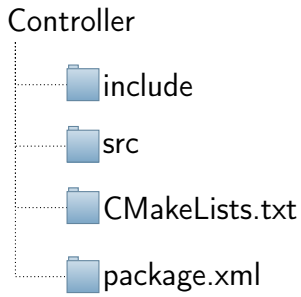

Figura 7.1: Árvore do diretório *Controller*.

control_strategies

A figura 7.2 ilustra o diretório *control_strategies*.

custom_plugins

A figura 7.3 ilustra o diretório *custom_plugins*.

Matlab

A figura 7.4 ilustra o diretório *Matlab*.

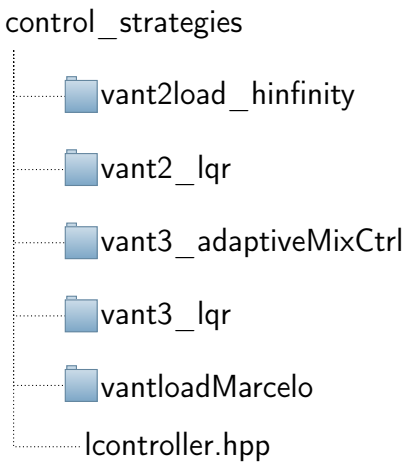

Figura 7.2: Árvore do diretório *control_strategies*.

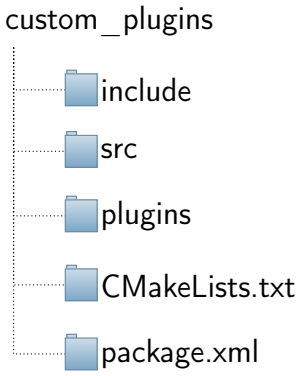

Figura 7.3: Árvore do diretório *custom_plugins*.

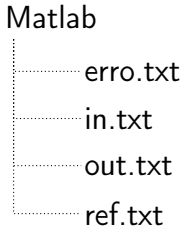

Figura 7.4: Árvore do diretório *Matlab*.

simulator_ msgs

A figura 7.4 ilustra o diretório *Matlab*.

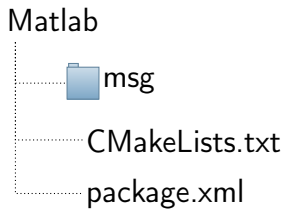

Figura 7.5: Árvore do diretório *simulador_ msgs*.

7.2 Arquivo config.xml

7.3 Arquitetura de Software

7.3.1 Tópicos e mensagens do ROS entre Controlador e Sensores/Plugins

Sensores/plugins com função sensorial comunicam com o Gazebo através de tópicos com mensagens padronizadas denominadas *Sensor*. Como estas mensagens podem chegar com ordens indefinidas, existe uma lógica no controlador que ordena uma estrtura denominada SensorArray de acordo com uma ordem previamente estabelecida pelo usuário através de um arquivo XML. Assim, quando o usuário projetar um novo controlador, ele terá acesso a um vetor de sensores conforme desejado. Isto está ilustrado na Figura 7.6.

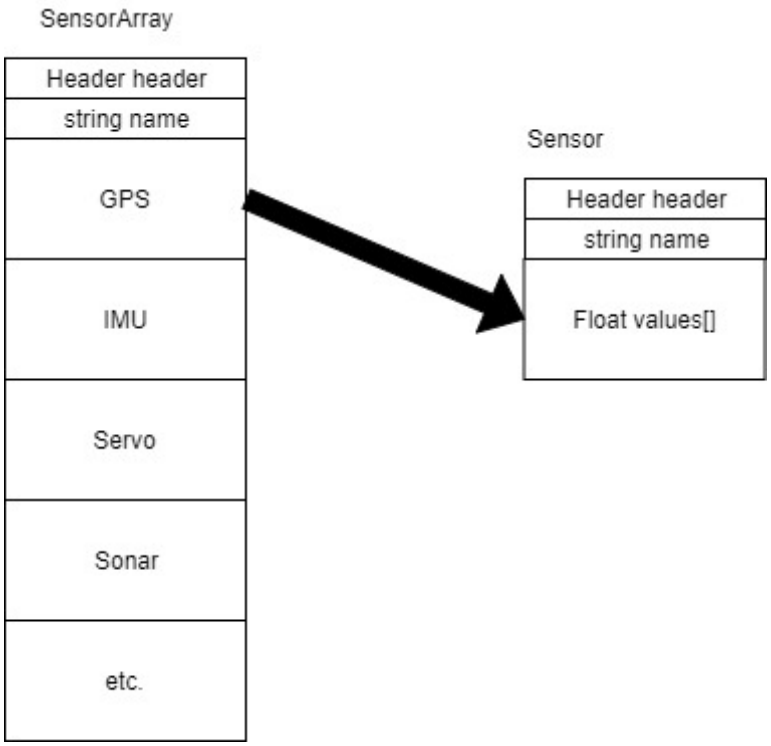

Figura 7.6: Mensagens utilizadas para comunicação entre Gazebo e Simulador

Além disso, a comunicação entre controlador e plugins ocorre por meio de tópicos com mensagens correspondendo a pontos flutuantes valores. Mais detalhes, o código se encontra auto-explicativo.

Capítulo 8

Interface Gráfica

8.1 Localização no projeto

8.2 Arquitetura de Software

Capítulo 9

Instalando e Compilando projeto

Este capítulo explica o processo de instalação e compilação do projeto. No entanto, para entender o assunto é necessário introduzir conhecimento prévio sobre sistema operacional Linux e a interface Shell. Além disso, é necessário conhecimento prévio de ROS e QT que, por sua vez, foram introduzidos nos capítulos x e y.

9.1 Introdução ao linux

Sistema operacional é o conjunto de programas que gerenciam recursos, processadores, dispositivos de entrada e saída e seus periféricos. Ele permite a comunicação entre o hardware e os demais softwares. Exemplos: Dos, Unix, Linux, Mac OS, OS-2, Windows NT. [Barreto,]

No contexto de plataformas Linux, o termo Linux denomina o núcleo do sistema operacional, responsável pela comunicação entre hardware e software do computador. No entanto, também é necessário outros programas para a interação entre usuário e kernel, por exemplo: compilador e a interface gráfica. Ou seja, no contexto de plataforma Linux, um sistema operacional é o conjunto do kernel e demais programas.

Existe diversos sistemas operacionais que utilizam o kernel Linux, diferenciando-se entre si apenas das ferramentas disponíveis para interação o usuário e kernel. Cada conjunto de ferramentas e kernel é denominado distribuição e os alguns exemplos são Debian, Ubuntu e openSUSE. Apesar da diferença, não se pode distinguir qual melhor distribuição, mas sim afirmar qual delas se adequa melhor à alguma tarefa de interesse.Mais informações pode ser obtidas em [de Paula,]

9.2 Shell

Uma das ferramentas disponíveis para a interação entre usuário e kernel é denominado Shell. Este é um programa criado com o intuito de interpretar comandos passados pelo usuário. Na verdade, ele é um arquivo executável, encarregado de interpretar comandos passados através de um conjunto de caracteres, transmití-los ao sistema e devolver resultados.

Quando se executa um comando, existe 3 tipos fluxos de informação, conforme é demonstrado na figura 9.1 e estão descritos a seguir

- **stdin**, do inglês "standard input", é o fluxo de dados de entrada. Por padrão, o stdin se refere ao teclado e é identificado pelo número 0;

- **stdout**, do inglês "standard output", é o fluxo de dados de saída. Por padrão, o stdout se refere à tela e é identificado pelo número 1;

- **stderr**, do inglês "standard error", é o fluxo de mensagens de erro. Por padrão, o stderr se refere à tela e é identificado pelo número 2:

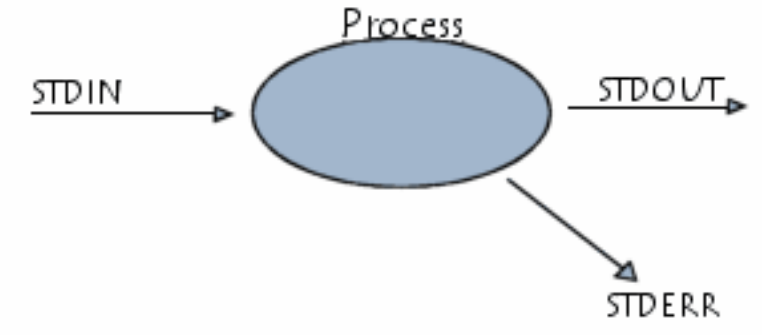

Figura 9.1: Fluxos de entrada e saída durante a execução de um comando. Imagem obtida de [CCM,]

Por padrão, quando se executa um programa, os dados são lidos a partir do teclado e o programa envia a sua saída e os seus erros para a tela. No entanto, também é possível ler os dados a partir de qualquer dispositivo de entrada, ou mesmo a partir de um arquivo, e enviar a saída para um dispositivo de visualização, arquivo etc. Mais informações podem ser obtidas em [CCM,]

9.3 Compilando e instalando projeto de software

O Shell, como já explicado, possui a função de executar comandos. Com a finalidade de automatizar a execução de um certo conjunto de comandos, existe disponível no ambiente Linux a ferramenta shell script. Segundo [Arruda,], "um shell script permite encadear comandos para solucionar tarefas mais complexas, como automatizar backups, redimensionar um lote de fotografias, limpar erros de um arquivo de texto, etc. Mas eles não são apenas uma simples sequência de comandos e podem usar características comuns às linguagens de programação, como condições (if-then-else) e até loops (for, repeat)".

Neste projeto, criou-se dois shell scripts denominados *build.sh* e *install.sh*. O primeiro, possui a funcionalidade de compilar todo o código fonte já descrito nos capítulos anteriores do ProVANT Simulator. Já o segundo, além de compilar o projeto, ele cria e registra variáveis de ambiente locais (serão descritas na próxima seção) e cria um link simbólico para fácil execução do sistema via terminal. Detalhes sobre os comandos utilizados nos scripts estão explicitados nos mesmos através de comentários.

9.4 Variáveis de ambiente

As variáveis de ambiente são espaços de memória responsáveis por armazenar informações pontuais do sistema. As variáveis podem ser variáveis locais ou variáveis globais. Os nomes das variáveis podem ser constituídos de quaisquer caracteres alfanuméricos ([Ferrari,]).

Um caso representativo da importância das variáveis de ambiente é o caso da variável de ambiente PATH. A variável PATH é utilizado muitas vezes para se registrar a localização de arquivos executáveis permitindo que o usuário e outros programas possam usufruir de suas funcionalidades.

No caso do ProVANT Simulator, utiliza-se as variáveis de ambiente do ROS e as variáveis de ambiente próprias criadas durante a instalação do sistema. Nesta seção será explicitada apenas sobre as variáveis de ambiente customizadas, sendo que detalhes sobre as variáveis de ambiente do ROS podem ser encontradas em na pagina `http://wiki.ros.org`.

As seguintes variáveis foram criadas:

- **PROVANT_ROS**: Diretório onde está localizado o código fonte do espaço de de trabalho do usuário no ROS
- **TILT_PROJECT**: Diretório onde está todo o projeto do ProVANT Simulator
- **TILT_STRATEGIES**: Diretório onde está localizado as bibliotecas dinâmicas compiladas com as estratégias de controle de ambiente de simulação
- **TILT_MATLAB** Diretório onde os arquivos com dados da simulação são armazenados
- **PROVANT_DATABASE**: Diretório onde se encontra os arquivos de descrição de modelos e cenários
- **GAZEBO_MODEL_PATH**: Este é uma exceção, não é uma variável ambiente criado, mas sim a atualização de uma variável ambiente do Gazebo. Aqui se encontram os modelos dos VANTs para serem adicionados em simulação.

Referências Bibliográficas

[Arruda,] Arruda, F. Introdução ao shell script: https://canaltech.com.br/linux/Introducao-ao-Shell-Script/, acessed july 10, 2018.

[Barreto,] Barreto, J. M. Sistema operacional: http://www.inf.ufsc.br/~j.barreto/cca/sisop/sisoperac.html, acessed july 10, 2018.

[CCM,] CCM. Linux – o shell: https://br.ccm.net/contents/320-linux-o-shell, acessed july 10, 2018.

[de Paula,] de Paula, F. B. O que é GNU/Linux: https://www.vivaolinux.com.br/linux/, acessed july 10, 2018.

[Ferrari,] Ferrari, F. Variáveis do shell: http://pcleon.if.ufrgs.br/~leon/Livro_3_ed/node41.html, acessed july 10, 2018.

Capítulo 10

Inicialização

Quando se utiliza o termo inicialização, dois significados aparecem: Inicialização da interface gráfica e inicialização da simulação propriamente dita. Este capítulo descreverá ambos, explicitando sobre o arquivos executáveis existentes no projeto e a maneira como são chamados. Inicialmente, será mostrado como é inicializado a interface gráfica e, posteriormente, como é inicializado Gazebo.

10.1 Inicializando interface gráfica

Para inicializar a interface gráfica, utiliza-se o comando via terminal de link simbólico criado durante a instalação do sistema.

A partir do comando:

```
provant_gui
```

será executado o arquivo executável **GUI** localizado na pasta $TILT_PROJECT/source/build.sh Este arquivo é obtido após a execução do script de compilação *install.sh* descrito no capítulo anterior.

10.2 Inicializando simulação

Após selecionado o cenário com seu respectivo VANT para simulação, é possível inicializar a simulação propriamente dita através do botão **Start Gazebo** localizado na tela inicial da interface gráfica como ilustrada na figura x.

Este botão executa uma configuração de um comando no terminal de acordo com o tipo de simulação desejada, depende da seleção (ou não) pelo usuário da opção de simulação via hardware-in-the-loop. Caso queira executar uma simulação "normal", será executado o comando **xterm e- roslaunch Database gazebo.lauch &**. Já caso queira realizar uma simulação via Hardware-in-the-loop, será executado o comando **xterm e- roslaunch Database hil.lauch &**.

O comando **xterm e-** equivale a abrir um outro terminal no emulador de terminais xterm e executar o comando passado como paramêtro. Já o comando roslaunch é

uma ferramenta utilizada para executar facilmente vários nós do ROS de acordo com a configuração prévia do arquivo launch passado como parâmetro.

10.2.1 Arquivos launch

Arquivos launch são arquivos XML desenvolvidos com o intuito de realizar seleção e configuração de

O primeiro executa o simulador gazebo e o nó controlador, já o segundo executará apenas o simulador gazebo. Na próxima seção será descrito cada um dos arquivos *launch* e dos arquivos executáveis envolvidos na simulação

Localização no projeto

Apêndice A

CMakeLists.txt

Esta seção foi extraída da página http://wiki.ros.org/catkin/CMakeLists.txt no dia 25/08/2017, visite-a para mais informações.

A.1 Visão geral e estrutura do arquivo CMakeLists.txt

O arquivo CMakeLists.txt armazena comandos de compilação e instalação de pacotes de software. Necessariamente, o arquivo **deve seguir formato e a ordem a seguir**.

1. Versão CMake necessária (cmake_minimum_required)
2. Nome do pacote (project())
3. Encontrar outros pacotes CMake/Catkin necessários para compilação (find_package())
4. Habilitação de suporte para módulos Python (catkin_python_setup())
5. Geradores de Mensagens/Serviços/Ações do ROS (add_message_files(), add_service_files(), add_action_files())
6. Invocar geração de Mensagem/Serviço/Ação (generate_messages())
7. Especificar pacote de compilação, informação e exportação (catkin_package())
8. Bibliotecas/Executáveis para compilação (add_library()/add_executable()/target_link_librar
9. Testes de construção (catkin_add_gtest())
10. Regras de instalação (install())

A.2 Versão CMake

Todo arquivo CMakeLists.txt deve começar com a declaração da versão do sistema. A versão requisitada é a 2.8.3 ou superior.

```
cmake_minimum_required(VERSION 2.8.3)
```

A.3 Nome do Pacote

O próximo item corresponde ao o nome do pacote do ROS. No exemplo a seguir, o pacote é chamado *robot_brain*.

```
project(robot_brain)
```

Obs.: Após esse comando, é possível fazer referência do nome do projeto em qualquer outro lugar através do uso da variável $PROJECT_NAME.

A.4 Encontrando dependências de pacotes CMake

É necessário especificar quais outros pacotes que precisam ser localizados para compilar o projeto. Essa especificação é realizada com o comando find_package e sempre há ao menos uma dependência pelo pacote catkin:

```
find_package(catkin REQUIRED)
```

Se o projeto depende de outros pacotes, eles são convertidos automaticamente em componentes do sistema catkin. Em vez de usar o comando find_package naqueles pacotes, é possível especificá-los como componentes para melhorar a legibilidade do script. O exemplo a seguir utiliza pacote 'nodelet" como componente.

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

Obs: É necessário usar somente o comando find_package para encontrar componentes. Não deve-se adicionar dependência de tempo de execução.

A.4.1 O find_package()

Se um pacote é localizado através do comando find_package, então resultará na criação de várias variáveis de ambiente do script CMakeLists que fornecem informação sobre o pacote encontrado. As variáveis de ambiente descrevem onde os arquivos dos pacotes exportados estão, quais bibliotecas o pacote depende e os caminhos destas bibliotecas. Os nomes seguem a convenção <PACKAGE NAME>_<PROPERTY>:

<NAME>_FOUND - configurado como verdadeiro caso a biblioteca é encontrada, caso contrário, falso
<NAME>_INCLUDE_DIRS ou <NAME>_INCLUDES - Caminhos de inclusão exportados pelo pacote
<NAME>_LIBRARIES ou <NAME>_LIBS - Bibliotecas exportadas pelo pacote
<NAME>_DEFINITIONS - Definições exportadas pelo pacote

A.4.2 Por que os pacotes são especificados como componentes?

Pacotes não são realmente componentes catkin. Em vez disso, a característica de componentes foi utilizada no desenvolvimento do script para economizar tempo de digitação significativo.

Para pacotes, será vantajoso utilizar o comando find_package para encontrá-los como componentes. Eles estão em um conjunto de variáveis criado com o prefixo catkin_. Por exemplo, quando você estiver usando um pacote denominado "nodelet" no seu código, sugere-se utilizar:

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

Isto significa que os caminhos incluídos, bibliotecas, etc. exportados pelo pacote "nodelet" são também anexadas às variáveis catkin_. Por exemplo, catkin_INCLUDE_DIRS contém os caminhos de inclusão não somente para pacote catkin mas também para para o pacote "nodelet".

Podemos de maneira alternativa achar o pacote "nodelet" com o comando:

```
find_package(nodelet)
```

Isto significa que os caminhos, bibliotecas e demais características do pacote "nodelet" não seriam adicionados às variáveis catkin_ . O que resulta em nodelet_INCLUDE_DIRS, nodelet_LIBRARIES, e etc.

As mesmas variáveis também são criadas usando

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

A.4.3 Boost

Caso esteja usando C++ e Boost, você necessita de invocar o comando find_package() para Boost e especificar quais aspectos da Boost que você está usando como componentes. Por exemplo, se você quiser usar threads da biblioteca Boost, você utilizaria o seguinte comando:

```
find_package(Boost REQUIRED COMPONENTS thread)
```

A.5 catkin_package()

O comando catkin_package() especifica informações do sistema catkin para o compilador.

Esta função deve ser utilizada antes de declarar quaisquer alvos com comandos add_library() ou add_executable(). A função tem 5 argumentos opcionais:

INCLUDE_DIRS - Caminhos de inclusão exportados pelo pacote
LIBRARIES - Bibliotecas exportadas do projeto
CATKIN_DEPENDS - Outros projetos catkin que este projeto depende
DEPENDS - Projetos não-catkin que este projeto depende
CFG_EXTRAS - Opções de configuração adicional

Observe o exemplo:

```
catkin_package(INCLUDE_DIRS include
LIBRARIES \${PROJECT_NAME}
CATKIN_DEPENDS roscpp nodelet
DEPENDS eigen opencv)
```

Isto indica que o diretório "include"dentro do diretório do pacote é o local que os cabeçalhos serão direcionados. A variável de ambiente $PROJECT_NAME avalia informação passada para a função project(), neste caso será ”robot_brain”. ”roscpp” e ”nodelet” são pacotes que necessitam estar presentes durante a compilação/execução deste pacote, já ”eigen” e ”opencv” são dependências do sistema que necessitam estar presentes para compilação/execução deste pacote.

A.6 Especificando alvos de compilação

Construir alvos pode ser realizadas de diversas formas, mas normalmente representam um das duas possibilidades:

Executable Target - programas a serem executados
Library Target - bibliotecas que podem ser usadas por alvos executáveis durante a compilação e/ou tempo de execução

A.6.1 Nomeando alvos

É muito importante notar que os nomes dos alvos de compilação no sistema catkin devem ser únicos sem considerar os diretórios que eles foram compilados/instalados. Este é um requisito do CMake. Entretanto, nomes únicos são apenas necessários internamente para CMake. Alguém pode obter um alvo renomeado como outro nome usando o comando set_target_properties():

Exemplo:

```
set_target_properties(rviz_image_view
PROPERTIES OUTPUT_NAME image_view
PREFIX "")
```

Isto irá trocar o nome do alvo rviz_image_view para image_view na compilação e instalação de saídas.

A.6.2 Diretório customizado de saída

Enquanto o diretório de saída padrão para executáveis e bibliotecas é comum a um valor razoável, ele deve ser personalizado em certos casos. Isto é, uma biblioteca contendo ligações de Python deve ser realocada em um diretório diferente a fim de ser importável no Python.:

Exemplo:

```
set_target_properties(python_module_library
PROPERTIES LIBRARY_OUTPUT_DIRECTORY ${CATKIN_DEVEL_PREFIX}${CATKIN_PACKAGE_PYTHON_DEST
```

A.6.3 Caminhos de inclusão e caminhos de bibliotecas

Antes de especificar alvos, você precisa especificar onde os recursos como arquivos de cabeçalho e bibliotecas podem ser localizados:

Caminhos de inclusão
Caminhos de biblioteca
include_directories(<dir1>, <dir2>, ..., <dirN>)
link_directories(<dir1>, <dir2>, ..., <dirN>)

a) include_directories()

O argumento para o comando include_directories deve ser as variáveis _INCLUDE_DIRS geradas pelo chamada do comando find_package e qualquer diretório adicional que necessita ser incluído. Se você estiver usando o sistema catkin e Boost, a chamada de include_directories() deve ser parecer:

```
include_directories(include ${Boost_INCLUDE_DIRS} ${catkin_INCLUDE_DIRS})
```

O primeiro argumento ”include” indica que o diretório include dentro do pacote faz parte também da caminho.

b) link_directories()

O comando link_directories() pode ser usada para adicionar novos caminhos de bibliotecas, no entanto, isto não é recomendado. Todos os pacotes catkin e CMake automaticamente tem sua informação de ligação adicionado quando são encontrados pelo find_package. Basta ligar as bibliotecas no target_link_libraries()
Exemplo:

```
link_directories(~/my_libs)
```

A.6.4 Alvos executáveis

Para especificar um alvo executável que deve ser construído, deve-se usar o comando CMake add_executable().

```
add_executable(myProgram src/main.cpp src/some_file.cpp src/another_file.cpp)
```

Isto irá construir um alvo executável denominado "myProgram"que, por sua vez, é constituídos de 3 arquivos fonte: src/main.cpp, src/some_file.cpp e src/another_file.cpp.

A.6.5 Alvos de bibliotecas

O comando CMake add_library() é usado para especificar bibliotecas para serem construídas. Por padrão o sistema catkin constrói bibliotecas dinâmicas.

```
add_library(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
```

A.6.6 target_link_libraries

Use o comando target_link_libraries() para especificar quais bibliotecas um alvo executável é ligado. Isto é feito tipicamente depois da chamada add_executable(). Adicione ${catkin_LIBRARIES} se o ROS não for encontrado.

Sintaxe:

```
target_link_libraries(<executableTargetName>, <lib1>, <lib2>, ... <libN>)
```

Exemplo:

```
add_executable(foo src/foo.cpp)
add_library(moo src/moo.cpp)
target_link_libraries(foo moo)  -- This links foo against libmoo.so
```

Observe que não há necessidade para uso de link_directories() na maioria dos casos, pois a informação é automaticamente enviada via find_package().

A.7 Mensagens alvos, Serviços alvos e Ações alvos

Arquivos de mensagens (.msg), serviços (.srv), e ações (.action) no ROS requerem um passo de construção especial antes de serem construídas e usadas por pacotes do ROS. O objetivo destas macros é gerar arquivos de linguagem específica de programação de maneira que alguém utilize mensagens, serviços e ações na linguagem de programação desejada. O sistema de compilação gerará ligações usando todos os geradores disponíveis (ex. gencpp, genpy, genlisp, etc).

Estas são três macros fornecidos para tratar de mensagens, serviços e ações respectivamente:

```
add_message_files
add_service_files
add_action_files
```

Estas macros devem ser seguidas por uma chamada que invoca a geração:

```
generate_messages()
```

Restrições/Pré-requisitos importantes

Estas macros devem vir antes do comando catkin_package() para correta geração.

```
find_package(catkin REQUIRED COMPONENTS ...)
add_message_files(...)
add_service_files(...)
add_action_files(...)
generate_messages(...)
catkin_package(...)
...
```

O comando catkin_package() deve ter dependência (CATKIN_DEPENDS) do pacote message_runtime.

```
catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)
```

Você deve usar find_package() para o pacote message_generation, seja sozinho ou como um componente do sistema catkin:

```
find_package(catkin REQUIRED COMPONENTS message_generation)
```

Seu arquivo package.xml deve conter uma dependência de construção do message_generation e uma dependência de tempo de execução message_runtime. Isso não é necessário se as dependências forem puxadas indiretamente de outros pacotes. Se você tem um alvo que (mesmo indiretamente) depende de algum outro alvo que precise de mensagens/serviços/ações a serem construídas, você precisa adicionar uma dependência explícita no alvo catkin_EXPORTED_TARGETS, para que eles sejam construídos na ordem correta. Este caso aplica-se quase sempre, a menos que seu pacote realmente não use nenhuma parte do ROS. Infelizmente, essa dependência não pode ser propagada automaticamente. (some_target correspondem ao nome do alvo definido por add _executable()):

```
add_dependencies(some_target ${catkin_EXPORTED_TARGETS})
```

Se você tem um pacote que cria mensagens e/ou serviços, bem como executáveis que usam estes, você precisa criar uma dependência explícita no alvo da mensagem gerada automaticamente para que eles sejam construídos na ordem correta. (some_target correspondem ao nome do alvo definido por add_executable ()):

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS})
```

Se o seu pacote satisfizer as duas condições acima, você precisa adicionar ambas as dependências, ou seja:

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_
```

A.7.1 Exemplo

Suponha que o seu pacote tenha duas mensagens em um diretório chamado "msg"chamadas "MyMessage1.msg"e "MyMessage2.msg"que dependem de std_msgs e sensor_msgs. Além disso, possui um serviço em um diretório chamado "srv"chamado "MyService.srv". O pacote define um executável que usa essas mensagens e serviços e um executável que usa algumas partes do ROS, mas não mensagens/serviços definidos neste pacote. Tal exemplo precisará do seguinte CMakeLists.txt:

```
# Obter informação sobre as dependências de compilação do pacote
find_package(catkin REQUIRED
COMPONENTS message_generation std_msgs sensor_msgs)

# Declaras arquivos mensagens para serem construídos
add_message_files(FILES
MyMessage1.msg
MyMessage2.msg
)

# Declara arquivos de serviços a serem construídos
add_service_files(FILES
MyService.srv
)

# Gera as mensagens e serviços com as linguagems específica de programação
generate_messages(DEPENDENCIES std_msgs sensor_msgs)

# Declara as dependências de tempo de execução do pacote
catkin_package(
CATKIN_DEPENDS message_runtime std_msgs sensor_msgs
)

# Define executáveis que usam mensagens.
add_executable(message_program src/main.cpp)
add_dependencies(message_program ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTE

# Define executável que não utiliza quaisquer mensagens/serviços fornecidos pelo paco
add_executable(does_not_use_local_messages_program src/main.cpp)
add_dependencies(does_not_use_local_messages_program ${catkin_EXPORTED_TARGETS})
```

Além disso, se você quer criar ações e tenha um arquivo de especificação de ação chamado "MyAction.action"no diretório "ação", você deve adicionar actionlib_msgs à lista de componentes que são encontrados com o sistema catkin e adicionar a seguinte chamada antes da chamada do comando generate_messages (...):

```
add_action_files(FILES
MyAction.action
)
```

Além disso, o pacote deve ter uma dependência de compilação em actionlib_msgs.

A.8 Habilitando suporte a módulos de Python

Se o seu pacote ROS fornecer alguns módulos Python, você deve criar um arquivo setup.py e chamar

```
catkin_python_setup()
```

Antes da chamada para generate_messages() e catkin_package().

A.9 Testes unitários

Há uma macro específica de catkin para manipulação de testes unitários baseados em gtest chamado catkin_add_gtest ().

```
catkin_add_gtest(myUnitTest test/utest.cpp)
```

A.10 Passo opcional: Especificando alvos instaláveis

Após o tempo de compilação, os alvos são colocados no espaço de desenvolvimento do espaço de trabalho catkin. No entanto, muitas vezes queremos instalar alvos no sistema para que eles possam ser usados por outros ou para uma pasta local para testar uma instalação no nível do sistema. Em outras palavras, se você quer ser capaz de fazer uma "instalação"do seu código, você precisa especificar onde os objetivos devem acabar.

Este é realizado usando o comando CMake install() que possui como argumentos:

TARGETS - alvos para instalar
ARCHIVE DESTINATION - bibliotecas estáticas e DLL (Windows) .stub
LIBRARY DESTINATION - bibliotecas dinâmicas não DLL e módulos
RUNTIME DESTINATION - Alvos executáveis e DLL (Windows) com estilo de bibliotecas compartilhadas

Observe o exemplo:

```
install(TARGETS ${PROJECT_NAME}
ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
```

Além desses destinos padrão, alguns arquivos devem ser instalados em pastas especiais. Isto é, uma biblioteca contendo módulos Python deve ser instalada em uma pasta diferente para ser importável no Python:

```
install(TARGETS python_module_library
ARCHIVE DESTINATION ${CATKIN_PACKAGE_PYTHON_DESTINATION}
LIBRARY DESTINATION ${CATKIN_PACKAGE_PYTHON_DESTINATION}
```

A.10.1 Instalando scripts executáveis de Python

Para o código Python, a regra de instalação parece diferente, pois não há uso das funções add_library() e add_executable(), de modo que o CMake determina quais arquivos são alvos e que tipo de alvos eles são. Em vez disso, use o seguinte em seu arquivo CMakeLists.txt:

```
catkin_install_python(PROGRAMS scripts/myscript
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

Informações detalhadas sobre a instalação de scripts e módulos de python, bem como as melhores práticas para layout de pastas podem ser encontradas no manual catkin.

Se você instala apenas scripts Python e não fornece módulos, não precisa criar o arquivo setup.py acima mencionado, nem chamar catkin_python_setup().

A.10.2 Instalando arquivos de cabeçalho

Os arquivos de cabeçalho também devem ser instalados na pasta "incluir", isso geralmente é feito instalando os arquivos de uma pasta inteira (opcionalmente filtrada por padrões de nomes de arquivos e excluindo subpastas SVN). Isso pode ser feito com uma regra de instalação que é a seguinte:

```
install(DIRECTORY include/${PROJECT_NAME}/
DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
PATTERN ".svn" EXCLUDE
```

Ou se a subpasta localizada na pasta incluir não corresponde ao nome do pacote:

```
install(DIRECTORY include/
DESTINATION ${CATKIN_GLOBAL_INCLUDE_DESTINATION}
PATTERN ".svn" EXCLUDE
```

Instalando arquivos roslaunch Files ou outros recursos

Outros recursos são arquivos launch que podem ser instalados em $CATKIN_PACKAGE_SHARE_DES

```
install(DIRECTORY launch/
DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}/launch
PATTERN ".svn" EXCLUDE)
```

Apêndice B

package.xml

Esta seção possui conteúdo retirado da página http://wiki.ros.org/catkin/package.xml no dia 25/08/2017. Para mais detalhes, visite-a.

B.1 Visão geral

O manifesto do pacote é um arquivo XML chamado package.xml que deve ser incluído com no diretório raiz de qualquer pacote compatível com catkin. Este arquivo define propriedades sobre o pacote, como o nome do pacote, números de versão, autores, responsáveis e dependências em outros pacotes catkin.

As dependências do pacote do sistema são declaradas em package.xml. Se eles estão faltando ou incorretos, é possível criar a partir da fonte e executar testes em sua própria máquina, mas o pacote não funcionará corretamente quando for lançado para a comunidade ROS. Outros dependem dessas informações para instalar o software necessário para usar seu pacote.

B.2 Formato 2 (Recomendado)

Este é o formato recomendado para novos pacotes. Também é recomendado que os pacotes anteriores do formato 1 sejam migrados para o formato 2. Para obter instruções sobre como migrar do formato 1 para o formato 2, consulte a página http://docs.ros.org/indigo/api/catkin/html/howto/format2/migrating_from_format_1.html#migrating-from-format1-to-format2 para mais informações.

A documentação completa para o formato 2 pode ser encontrada em http://www.ros.org/reps/rep-0140.html.

B.2.1 Estrutura básica

Cada arquivo package.xml tem uma tag <package> como tag raíz do documento.

```
<package format="2">

</package>
```

B.2.2 Tags requisitadas

Há um conjunto mínimo de tags que necessitam de ser alocadas dentro da tag <package> para tornar o manifesto do pacote completo.

<name> - O nome do pacote
<version> - O número da versão do pacote (obrigatório ter 3 inteiros separados por pontos)
<description> - Descrição do conteúdo do pacote
<maintainer> - Nome das pessoas que realizam a manutençaõ do pacote
<license> - A(s) licença(s) de software (por exemplo, GPL, BSD, ASL) sob a qual o código é liberado.

Observe um exemplo de manifesto para um pacote fictício denominado foo_core.

```
<package format="2">
<name>foo_core</name>
<version>1.2.4</version>
<description>
This package provides foo capability.
</description>
<maintainer email="ivana@osrf.org">Ivana Bildbotz</maintainer>
<license>BSD</license>
</package>
```

B.2.3 Dependências

O manifesto do pacote com tags mínimas não especifica nenhuma dependência em outros pacotes. Os pacotes podem ter seis tipos de dependências:

- **Dependências de compilação** especifica quais pacotes são necessários para compilar este pacote. Este é o caso quando algum arquivo destes pacotes é requisitado em tempo de compilação. Isto pode ser realizado incluindo cabeçalhos destes pacotes em tempo de compilação, ligando bibliotecas destes pacotes ou requisitando algum outro recurso em tempo de compilação (especialmente quando estes pacotes são encontrados por find_package() no CMake). Num cenário de cross-compilação construir dependências são próprias para uma arquitetura segmentada.
- **Dependências de exportação** especifica quais pacotes são necessários para criar bibliotecas. Este é o caso quando você inclui indiretamente seus cabeçalhos em cabeçalhos públicos neste pacote (especialmente quando estes pacotes são declarados como (CATKIN_)DEPENDs em catkin_package() no CMake).
- **Dependências de execução** Especifica quais pacotes são necessários para executar o código neste pacote. Este é o caso quando você depende de bibliotecas compartilhadas neste pacote (especialmente quando esses pacotes são declarados como (CATKIN_)DEPENDs em catkin_package() no CMake).

- **Dependências de testes** Especifica apenas dependências adicionais para testes unitários. Nunca devem duplicar quaisquer dependências já mencionadas como dependências de compilação ou execução.
- **Dependências de ferramentas de construção** Especifica as ferramentas do sistema de compilação que este pacote precisa construir. Normalmente, a única ferramenta de compilação necessária é o sistema catkin. Em um cenário de compilação cruzada, as dependências da ferramenta de compilação são para a arquitetura na qual a compilação é executada.
- **Dependências de ferramentas de documentação** especifica ferramentas que esse pacote precisa para gerar documentação.

Esses seis tipos de dependências são especificados usando as seguintes tags respectivas:

<depend> especifica que uma dependência é uma dependência de compilação, exportação e execução. Esta é a etiqueta de dependência mais usada.
<buildtool_depend>
<build_depend>
<build_export_depend>
<exec_depend>
<test_depend>
<doc_depend>

Todos os pacotes possuem pelo menos uma dependência. A seguir está um exemplo de uma dependência de ferramenta de compilação no sistema catkin.

```
<package>
<name>foo_core</name>
<version>1.2.4</version>
<description>
This package provides foo capability.
</description>
<maintainer email="ivana@osrf.org">Ivana Bildbotz</maintainer>
<license>BSD</license>
<buildtool_depend>catkin</buildtool_depend>
</package>
```

Um exemplo mais realista que especifica as dependências de compilação, exec, teste e doc pode parecer o seguinte.

```
<package>
<name>foo_core</name>
<version>1.2.4</version>
<description>
This package provides foo capability.
</description>
<maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
```

```
<license>BSD</license>
<url>http://ros.org/wiki/foo_core</url>
<author>Ivana Bildbotz</author>
<buildtool_depend>catkin</buildtool_depend>
<depend>roscpp</depend>
<depend>std_msgs</depend>
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
<exec_depend>rospy</exec_depend>
<test_depend>python-mock</test_depend>
<doc_depend>doxygen</doc_depend>
</package>
```

B.2.4 Metapackages

Muitas vezes, é conveniente agrupar vários pacotes como um único pacote lógico. Isso pode ser conseguido através de metapackages. Um metapackage é um pacote normal com a seguinte tag de exportação no package.xml:

```
<export>
<metapackage />
</export>
```

Além de uma dependência <buildtool_depends> necessária, os metapackages só podem ter dependências executadas em pacotes dos quais eles agrupam.

Além disso, um metapackage possui um arquivo CMakeLists.txt obrigatório, exigido:

```
cmake_minimum_required(VERSION 2.8.3)
project(<PACKAGE_NAME>)
find_package(catkin REQUIRED)
```

Note: substitua <PACKAGE_NAME> com o nome do metapackage.

B.2.5 Tags adicionais

<url> - Um URL para obter informações sobre o pacote, normalmente uma página do wiki no ros.org.
<author> - O(s) autores do pacote

B.3 Formato 1 (legado)

Os pacotes catkin mais antigos usam o formato 1. Se a tag <package> não tiver um atributo de formato, é um pacote de formato 1. Use o formato 2 para novos pacotes.

B.3.1 Estrutura básica

Cada arquivo package.xml tem a tag <package> como tag raís no documento.

```
<package>

</package>
```

B.3.2 Tags Necessárias

Há um conjunto mínimo de tags que precisam ser aninhadas dentro da tag <package> para tornar o pacote manifesto completo.

<name> - O nome do pacote
<version> - A versãodo pacote (Obrigatório ter 3 inteiros separados por pontos)
<description> - Uma descrição do conteúdo do pacote
<maintainer> - O(s) nome das pessoa(s) responsáveis pela manutenção do pacote
<license> - A(s) licença(s) de software (por exemplo, GPL, BSD, ASL) sob a qual o código é liberado.

Como exemplo, aqui está o pacote manifesto para um pacote de ficção chamado foo_core.

```
<package>
<name>foo_core</name>
<version>1.2.4</version>
<description>
This package provides foo capability.
</description>
<maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
<license>BSD</license>
</package>
```

B.3.3 Dependências de compilação, execução e testes

O manifesto do pacote com tags mínimas não especifica nenhuma dependência em outros pacotes. Pacotes podem ter quatro tipos de dependências:

- **Dependências de ferramentas de construção** especifica as ferramentas do sistema de compilação que este pacote precisa construir. Normalmente, a única ferramenta de compilação necessária é o sistema catkin. Em um cenário de compilação cruzada, as dependências da ferramenta de compilação são para a arquitetura na qual a compilação é executada.
- **Dependências de construção** especifica quais pacotes são necessários para criar este pacote. Este é o caso quando um arquivo desses pacotes é necessário no momento da compilação. Isso pode incluir cabeçalhos desses pacotes no momento da compilação, vinculando as bibliotecas desses pacotes ou exigindo

qualquer outro recurso em tempo de compilação (especialmente quando esses pacotes são find_package()-ed no CMake). Em um cenário de compilação cruzada, as dependências de compilação são para a arquitetura segmentada.

- **Dependências de execução** Especifica quais pacotes são necessários para executar o código neste pacote ou criar bibliotecas neste pacote. Este é o caso quando você depende de bibliotecas compartilhadas ou inclui transitivamente seus cabeçalhos em cabeçalhos públicos (especialmente quando estes pacotes são declarados como (CATKIN_)DEPENDs em catkin_package() no CMake).
- **Dependências de teste** Especifica apenas dependências adicionais para testes unitários. Eles nunca devem duplicar quaisquer dependências já mencionadas como dependências de compilação ou execução.

Esses quatro tipos de dependências são especificados usando as seguintes tags respectivas:

- <buildtool_depend>
- <build_depend>
- <run_depend>
- <test_depend>

Todos os pacotes possuem pelo menos uma dependência, uma dependência de ferramenta de compilação no catkin conforme mostra o exemplo a seguir.

```
<package>
<name>foo_core</name>
<version>1.2.4</version>
<description>
This package provides foo capability.
</description>
<maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
<license>BSD</license>
<buildtool_depend>catkin</buildtool_depend>
</package>
```

Um exemplo mais realista que especifica as dependências de compilação, tempo de execução e teste pode ser o seguinte.

```
<package>
<name>foo_core</name>
<version>1.2.4</version>
<description>
This package provides foo capability.
</description>
<maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
<license>BSD</license>
<url>http://ros.org/wiki/foo_core</url>
<author>Ivana Bildbotz</author>
```

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>message_generation</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>message_runtime</run_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>
<test_depend>python-mock</test_depend>
</package>
```

B.3.4 Metapackages

Muitas vezes, é conveniente agrupar vários pacotes como um único pacote lógico. Isso pode ser conseguido através de metapackages. Um metapackage é um pacote normal com a seguinte tag de exportação no package.xml:

```
<export>
<metapackage />
</export>
```

Além de uma dependência <buildtool_depends> necessária, os metapackages só podem ter dependências executadas em pacotes dos quais eles agrupam.

Além disso, um metapackage possui um arquivo CMakeLists.txt obrigatório, exigido:

```
cmake_minimum_required(VERSION 2.8.3)
project(<PACKAGE_NAME>)
find_package(catkin REQUIRED)
catkin_metapackage()
```

Note: substitua <PACKAGE_NAME> com o nome do metapackage.

B.3.5 Tags Adicionais

- <url> - Um URL para obter informações sobre o pacote, normalmente uma página do wiki no ros.org.
- <author> - O(s) autor(es) do pacote

Referências Bibliográficas

[Arruda,] Arruda, F. Introdução ao shell script: https://canaltech.com.br/linux/Introducao-ao-Shell-Script/, acessed july 10, 2018.

[Barreto,] Barreto, J. M. Sistema operacional: http://www.inf.ufsc.br/~j.barreto/cca/sisop/sisoperac.html, acessed july 10, 2018.

[CCM,] CCM. Linux – o shell: https://br.ccm.net/contents/320-linux-o-shell, acessed july 10, 2018.

[de Paula,] de Paula, F. B. O que é GNU/Linux: https://www.vivaolinux.com.br/linux/, acessed july 10, 2018.

[Ferrari,] Ferrari, F. Variáveis do shell: http://pcleon.if.ufrgs.br/~leon/Livro_3_ed/node41.html, acessed july 10, 2018.